

AD-A091 015

MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/6 9/2

A SEMANTICS OF SYNCHRONIZATION.(U)

SEP 80 C R SEAQUIST

MIT/LCS/TM-176

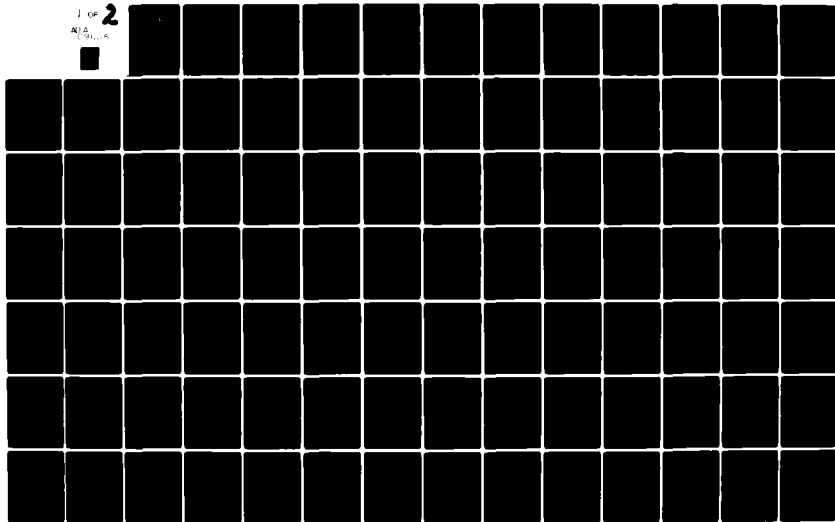
N00014-75-C-0661

NL

UNCLASSIFIED

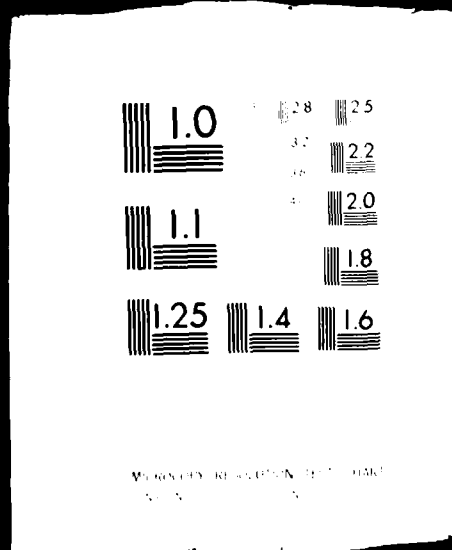
1 OF 2

ALIA
C91114



1 OF 2

AD. A
091015



LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

12

LEVEL II

MIT/LCS/TM-176

A SEMANTICS OF SYNCHRONIZATION

Carl R. Seaquist

DTIC
ELECTE
OCT 30 1980
S D
E

September 1980

This research was supported in part by the Advanced Research
Projects Agency of the Department of Defense monitored
by the Office of Naval Research under contract N00014-75-C-0661
and in part by the National Science Foundation under
grant MCS78-17698

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

80 10 23 016

AD A091015

DDC FILE COPY

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)


⑨ Master's thesis

14 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM	
1. REPORT NUMBER MIT/LCS/TM-176		2. GOVT ACCESSION NO. AD-A091015	
3. TITLE (and Subtitle) A Semantics of Synchronization		3. RECIPIENT'S CATALOG NUMBER	
4. AUTHOR(s) Carl R. Seaquist		5. TYPE OF REPORT & PERIOD COVERED M.S. Thesis-August 1980	
5. PERFORMING ORGANIZATION NAME AND ADDRESS MIT/Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139		6. PERFORMING ORG. REPORT NUMBER MIT/LCS/TM-176	
6. CONTROLLING OFFICE NAME AND ADDRESS NSF/Associate ARPA/Department of Defense / Program Director 1400 Wilson Boulevard / Office Computing Act. Arlington, VA 22209 / Washington, D.C. 20550		7. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0661 NSF-MCS78-17698	
7. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR/Department of the Navy Information Systems Program Arlington, VA 22217		8. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
8. DISTRIBUTION STATEMENT (of this Report) This document has been approved for public release and sale; its distribution is unlimited		9. REPORT DATE September 1980	
9. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		10. NUMBER OF PAGES 113	
10. SUPPLEMENTARY NOTES		11. SECURITY CLASS. (of this report) Unclassified	
11. KEY WORDS (Continue on reverse side if necessary and identify by block number)		12. DECLASSIFICATION/DOWNGRADING SCHEDULE	
12. ABSTRACT (Continue on reverse side if necessary and identify by block number) This paper presents a rigorous framework in which to discuss the synchronization necessary to coordinate accesses to a resource. The framework, among other things, provides a method for specifying concurrency and forms the semantic basis of a synchronization mechanism which avoids certain unfortunate characteristics of monitors and serializers. Synchronization is viewed as being managed by a resource			

409648 9m

20.

guardian. A synchronization problem is defined as a predicate on event sequences. The interaction of a guardian and the rest of the system is formalized in terms of a two person game. This formalization results in precise definitions of guardian and guardian behavior. The notion of a "good" or optimal solution is defined, and the solutions to certain classes of synchronization problems are characterized. An abstract description of the general actions of a guardian is given. This general description, with some restrictions, forms the basis of a simple synchronization mechanism for actually implementing solutions. The mechanism is given a rigorous semantics based on the definition of guardian. This facilitates the verification of correctness. Many examples of the use of the mechanism are given and its advantages are discussed.



A Semantics of Synchronization

by

Carl R. Seaquist

26 August 1980

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DDC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist.	Avail and/or special
A	

© Massachusetts Institute of Technology 1980

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract N00014-75-C-0661 and in part by the National Science Foundation under grant MCS78-17698.

Abstract

This paper presents a rigorous framework in which to discuss the synchronization necessary to coordinate accesses to a resource. The framework, among other things, provides a method for specifying concurrency and forms the semantic basis of a synchronization mechanism which avoids certain unfortunate characteristics of monitors and serializers. Synchronization is viewed as being managed by a resource guardian. A synchronization problem is defined as a predicate on event sequences. The interaction of a guardian and the rest of the system is formalized in terms of a two person game. This formalization results in precise definitions of guardian and guardian behavior. The notion of a "good" or optimal solution is defined, and the solutions to certain classes of synchronization problems are characterized. An abstract description of the general actions of a guardian is given. This general description, with some restrictions, forms the basis of a simple synchronization mechanism for actually implementing solutions. The mechanism is given a rigorous semantics based on the definition of guardian. This facilitates the verification of correctness. Many examples of the use of the mechanism are given and its advantages are discussed.

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 1980 in partial fulfillment of the requirements for
the Degree of Master of Science

Acknowledgments

I would like especially to thank my thesis advisor, Professor Irene Greif, for her support and encouragement in pursuing this research. I am also thankful to Deepak Kapur, Bill Weihl, Allen Emerson, Will Clinger, and Joyce Mahan for reading various drafts of the paper and for making many illuminating and helpful comments. I am also indebted to many friends for their patience and kindness. Deserving special mention are Deepak Kapur, M. K. Srivas, Glen Miranker, and Louie Skipper.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract N00014-75-C-0661 and in part by the National Science Foundation under grant MCS78-17698.

CONTENTS

1. Introduction	7
1.1 Related Work	8
1.2 Outline of Paper	9
2. Guardians and Guardian Behavior	11
2.1 Resource Guardian	11
2.2 Event Sequences and Predicates	13
2.2.1 Events and Event Sequences	13
2.2.2 Predicates on Event Sequences	17
2.3 General Polling Guardian	20
2.4 Examples of General Polling Guardians	23
2.5 Simple Polling Guardian	28
2.6 Expression of Solutions as Simple Guardians	32
3. Guardian/Environment Game	35
3.1 Definitions	35
3.2 Specification of Concurrency	38
3.3 Continuity of Specifications	43
3.4 Simple Predicates	46
4. A Synchronization Mechanism	50
4.1 The Synchronization Mechanism	51
4.1.1 Overview	51
4.1.2 Definition of the Synchronization Mechanism	55
4.1.3 Implementation Issues	64
4.2 Readers/Writers Problem	65
4.3 Disk Scheduler Problem	73
4.4 Five Dining Philosophers	77
4.5 Bounded Buffer Problem	82
4.6 Proof of Implementation Correctness	86

4.6.1 An Outline of the Methodology	86
4.6.2 Correctness of a Solution to the Readers/Writers Problem	88
4.6.3 Correctness of a Bounded Buffer Solution	94
4.6.4 Remarks on the Methodology	97
4.7 Conclusions	99
 5. Summary and Directions for Future Work	 103
5.1 Summary	103
5.2 Future Research	105
 6. References	 108

FIGURES

Fig. 2.2.1.1. Examples of Events and Functions on Events	15
Fig. 2.2.1.2. Sequence Notation and Examples of Event Sequences: Legal and Illegal	17
Fig. 2.3.3. Scheme for a General Polling Guardian	21
Fig. 2.3.4. A Solution to $Mx(r,r')$	22
Fig. 2.4.5. A Semaphore Solution to $Mx(r,r')$	24
Fig. 2.4.6. Weak Semaphore Solution to $Mx(r,r')$	25
Fig. 2.4.7. Monitor Solution to Readers Priority Version of Readers/Writers Problem	27
Fig. 2.4.8. A Solution Equivalent to the Monitor Solution	28
Fig. 2.5.9. Scheme for Simple Polling Guardian	29
Fig. 2.5.10. Simple Polling Guardian Solution to Readers/Writers Problem	31
Fig. 2.6.11. Subsets of Requests	33
Fig. 2.6.12. Specification of a Solution to Readers/Writers Problem	34
Fig. 3.1.1. An Instance of Guardian/Environment Game Defined by $P \equiv Mx(o,o)$	36
Fig. 3.1.2. Sequences and Their Corresponding Expanded Versions	39
Fig. 3.2.3. One at a Time Solution to Readers/Writers Problem	39
Fig. 3.2.4. Greedy and Lazy Solutions to $Mx(r,r')$	42
Fig. 4.1.1.1. Simple Polling Guardian Procedure Solving $Mx(r,r')$	52
Fig. 4.1.1.2. Differences in a Standard and Modified Invocation	52
Fig. 4.1.2.3. Summary of Protector's Operation	59
Fig. 4.1.2.4. Semantics of a Protector	60
Fig. 4.1.2.5. Synchronization Type for Mutual Exclusion	61
Fig. 4.2.6. FCFS Solution to Readers/Writers Problem	66
Fig. 4.2.7. Weak Readers Priority Solution to the Readers/Writers Problem	68
Fig. 4.2.8. Readers Priority Solution to the Readers/Writers Problem	69
Fig. 4.2.9. Writers Priority Solution to the Readers/Writers Problem	70
Fig. 4.2.10. Fair Solution to the Readers Priority Readers/Writers Problem	72
Fig. 4.3.11. The SCAN Solution to the Disk Scheduler Problem	74
Fig. 4.3.12. The CSCAN Solution to the Disk Scheduler Problem	76
Fig. 4.4.13. The Dining Philosophers Problem	77
Fig. 4.4.14. Optimal Solution to the Dining Philosophers Problem	78
Fig. 4.4.15. FCFS Solution to the Dining Philosophers Problem	79
Fig. 4.4.16. Fair Solution to Dining Philosophers Problem with Very Hungry Sages	81
Fig. 4.5.17. Nearly FIFO Solution to the Bounded Buffer Problem	83
Fig. 4.5.18. Solution to the Bounded Buffer Problem	85
Fig. 4.6.2.19. Fair Solution to the Readers Priority Readers/Writers Problem	90
Fig. 4.6.3.20. Solution to the Bounded Buffer Problem	95
Fig. 4.6.4.21. Weak Readers Priority Solution to the Readers/Writers Problem	98
Fig. 4.7.22. A Procedure for Testing a Synchronization Type s	102

1. Introduction

The purpose of this paper is to give a rigorous framework in which to discuss modular mechanisms of synchronization and, using this framework, to develop a programming construct for implementing solutions to synchronization problems. Attention is confined to guardian-type synchronization mechanisms [LAVE78] -- i.e., well-structured modular mechanisms which are resource protection envelopes capable of realizing both exclusion and priority constraints. The behavior of synchronization mechanisms is defined in terms of sets of possible sequences of events. The interaction between the synchronization mechanism and its environment is made explicit in terms of a two person game. The developed framework permits the pursuit of the following goals:

- i) To design a versatile yet semantically simple synchronization programming construct.
- ii) To address the question of accurate problem specification.
- iii) To examine the semantics of synchronization mechanisms that are currently described in the literature.
- iv) To define a notation for abstractly describing the solutions to synchronization problems.

Besides being of technical interest, the pursuit of the above objectives enhances the understanding of the nature of synchronization problems and the semantics and limitations of certain modular synchronization mechanisms. The programming construct which is introduced is demonstrated to be a useful tool which aids in the speedy implementation of efficient and correct solutions to the synchronization problems which arise in a multi-processing environment. To summarize, then, the main contributions of this paper are a method for viewing synchronization and a framework which results in a versatile synchronization mechanism. In addition, the framework provides an approach to problem specification and the basis for verification

of solutions.

1.1 Related Work

Since the late 1960's much has been written on the synchronization and coordination of processes. This research is particularly indebted to the work on message passing by Hewitt [HEW177], work on specification and on event semantics by Greif [GRE175], and to Dijkstra's Secretary metaphor [DIJK71].

Works by Greif [GRE175], [GRE177] and by Lavenhal [LAVE78] develop the resource guardian model which they used to derive a method for precisely specifying synchronization problems. In this paper the notion of resource guardian (i.e., an envelope surrounding a resource) is combined with Dijkstra's Secretary metaphor [DIJK71] in order to obtain a scheme which can be used to describe the important aspects of synchronization. In each of [GRE175], [GRE177], and [LAVE78], emphasis is placed on specifying exclusion and priority constraints with little or no mention of concurrency constraints which specify that a certain amount of parallelism is desired in a solution. For example, consider a typical specification of one of the simplest versions of the readers/writers problem, [GRE177], [LAVE78], [BLOO79]. A guardian which allows only one read or write in the database at a time satisfies this specification. Unfortunately this implementation violates our concept of a good solution to the problem which is that reads should be able to proceed in parallel. In this paper a method for writing specifications that exclude this unfortunate solution is presented.

The idea of applying game theory in order to make explicit certain situations which arise in a multi-processing system has also appeared recently in other work. In particular, Ladner [LADN79] describes the interaction of a process and a system in terms of a game: The process wins if it can enter a state in which it is locked out (from, say, a resource or a section of code) while the system wins if it can prevent the lockout. Others, including Devillers [DEV177] and Reif and Peterson [REIF80], have used games to define other system properties. Reif and Peterson use a "game-like

semantics" to develop a paradigm for viewing general multi-processing problems.

The programming mechanism described in this paper has grown out of work done on monitors [HOAR74] and serializers [HEW179a], [HEW179b]. It combines many of the good points of each of these constructs while avoiding some of their pitfalls. Monitor solutions are sometimes difficult to understand because the locus of control is very unstructured. In addition, as will be shown later, both serializers and monitors exhibit undesirable behavior in some situations because of the way they have been designed. All these problems are avoided in the programming mechanism presented here. Work by Bloom [BLOO79] on ways of evaluating synchronization constructs is used as a guide in developing this particular programming mechanism. Recent work by Hewitt [HEW179b] is very similar in spirit to the work described in this paper but his construct seems to lack the firm semantic foundation of the construct developed here. The dedication of a process to do conceptual polling in the mechanism developed here seems to simplify control without necessarily sacrificing efficiency. In a recent paper [HANS78], Brinch Hansen describes a synchronization technique which also makes use of a dedicated process to accomplish synchronization. He does not, however, treat events as data objects and thus loses some of the debugging advantages which are described in Chapter 4. In [LAVE78], Lavalenthal describes a technique for the synthesis of synchronization code from problem specifications. Unfortunately, because of limitations in his solution specification language, his method cannot handle several important cases. It is shown that the design of our construct is general enough to permit handling of these cases.

1.2 Outline of Paper

The paper begins with an informal introduction which motivates our view of synchronization. Then this view is made more formal to provide a base for careful reasoning about synchronization. Finally a practical programming construct for synchronizing activity is presented.

In the second chapter the notions of resource guardian, guardian lifetime, and guardian behavior are introduced informally. Events and event sequences are defined, making it possible to discuss predicates on event sequences and to introduce some abbreviations for expressing commonly used predicates. Next a particular approach for viewing resource guardians is presented. This results in the introduction of the notions of general and simple polling guardians. The last section describes a notation for abstractly defining simple polling guardians.

The third chapter describes the interaction between a resource guardian and the rest of the system (i.e., the guardian's environment) in terms of a two person infinite game with perfect information [GALE53]. A guardian is then formally defined as any functional strategy for the second player. Also addressed in this chapter is the problem of how to specify that a solution must permit concurrency whenever possible. In the last two sections continuous and simple predicates are defined, and several theorems are proved which help characterize solutions to synchronization problems. These characterizations are useful in verifying the correctness of solutions.

The fourth chapter uses the results from the preceding two chapters and defines a programming construct which represents a practical approach to implementing solutions to synchronization problems. The chapter begins with a complete description of the programming construct and goes on to consider many examples of classical synchronization problems. The treatment of these problems is different from the usual in that many solutions of diverse style and behavior are given to each of the more interesting problems. An approach to verifying the correctness of implementations is discussed, and several interesting examples are examined.

The fifth chapter summarizes the results of the paper and points to areas where further research might profitably be pursued.

2. Guardians and Guardian Behavior

In this chapter the terms *resource guardian*, *guardian behavior*, *behavior equivalence*, *problem specification*, and *problem solution* are defined informally. *Events*, *event sequences*, and *predicates on event sequences* are defined. Then several abbreviations are introduced which make the expression of many common predicates quite simple. A framework for viewing resource guardians is proposed which results in what will be called a polling guardian. The generality of this approach will be demonstrated by describing solutions to certain well known synchronization problems in terms of the polling guardian. These solutions will be shown to be behaviorally equivalent to other well accepted solutions implemented in terms of the more common synchronization mechanisms found in the literature. Next a simplified version of the polling guardian is examined. Although there is certain behavior which such a guardian cannot enforce, it will be argued that this simplified version is not only *adequate but desirable*. Finally notation for abstractly specifying a solution is described.

2.1 Resource Guardian

The term *resource guardian* (or simply *guardian*) is used to refer to synchronization code which monitors resource activity and uses this information to control access to the resource. To access the resource, a process must make a request to the guardian. A request event is associated with the guardian's receipt of this request. When the guardian decides that it is proper for the process to access the resource, the guardian signals the process that it can enter (i.e., access) the resource. Associated with the guardian's giving of this signal is an *enter* event. Finally when the process finishes with the resource, it notifies the guardian that it has exited from the resource. An *exit* event is associated with the guardian's receipt of this notification. Three event types are important in considering a guardian and its interaction with a particular access by a process. These are the *request* for service event, the *enter* or grant of service event, and the *exit* or termination of service event.

Notice that all events are occurrences as seen by the guardian. In particular, the enter event is associated with the sending of the signal to the process, *not* with the process's receipt of the signal.¹ Also note that when a process makes a request of a guardian, it will make no further requests of this guardian until it exits from the resource. For an example of a resource guardian, consider a critical section of code which is protected by a semaphore. The critical section of code corresponds to the resource, and the P-instruction and the V-instruction make up the guardian. When a process wants to enter the critical section, it begins to execute the P-instruction. The request event occurs at the start of execution of the P-instruction. Eventually it will be decided which of the processes currently engaged in executing the P-instruction (i.e., waiting) will be allowed to finish with the instruction and permitted to enter the critical section. The enter event occurs when the guardian decides that the process has finished executing the P-instruction. When the process executes the V-instruction, it signals that it is exiting the critical section by decrementing the semaphore. Thus the exit event occurs the instant that the decrementing of the semaphore is complete.

The events associated with a guardian are assumed to be totally ordered by their time of occurrence. Practically, this means there must be some sort of arbiter which serializes the guardian events which might otherwise appear to occur simultaneously. The necessity of arbitration is no surprise since simultaneous requests must be arbitrated in order to implement monitors or serializers. It is also assumed that the events associated with a guardian form a sequence. This means, first, that between any two events there are only finitely many events -- i.e., events are not "dense" [SCHW78] -- and, second, that there is a first event. With these assumptions understood, the following informal definitions can be made. More formal definitions of many of these terms can be found in 3.1.

1. This is different from the way in which Carl Hewitt [HEW177] associates events in message passing systems, where events are always associated with the receipt of the message.

A *lifetime* of a guardian is a sequence (finite or infinite) of events which might be the result of the operation of the guardian. The *behavior* of a guardian G is the set of all lifetimes of G , and is denoted by $B(G)$. Note that many different lifetimes are possible due to the different orderings of requests and exits which can occur. Two guardians are *equivalent* if they have identical behaviors. A synchronization *problem specification* is any predicate defined on sequences of guardian events. The *behavior set* of a problem consists of all the sequences that satisfy the problem specification. The behavior set of a predicate P also is denoted by P . This should cause no confusion. A guardian *satisfies* a problem specification, or solves the problem, if and only if its behavior is a subset of the problem's behavior set. A *solution* to a problem specification is any guardian which satisfies the specification.

2.2 Event Sequences and Predicates

In this section events and event sequences are defined, and many examples are given. Then predicates on event sequences are discussed. Several useful abbreviations for expressing common predicates are introduced.

2.2.1 Events and Event Sequences

A guardian event can be one of three types: request, enter, or exit. Associated with each event is an operation which will be (or has been) performed on the resource and, also, a process which will perform (or has performed) the operation. Thus each event can be viewed as an ordered triple consisting of type, operation, and process along with some sort of unique identification associated with a particular access attempt. In this way it is possible to have two distinct request events by the same process to perform the same operation. Two such requests are distinct because they occur on different accesses.

Request events are denoted by r either subscripted or primed. Subscripts and primes are used to make explicit the uniqueness of access. Request event variables are denoted by r , either subscripted or primed to distinguish among variables. A variable r ranges over all possible requests. Often it is necessary to denote a request or request variable with an explicit operation and/or process. This is done by writing the request or request variable followed by the operation and/or process id enclosed in brackets. To avoid ambiguity, numerals are used to denote process ids but never operations. A variable $r[op]$ would range over all requests with the operation op . The function o (read "the operation of") takes a request as an argument and returns the operation to be performed. The function p (read "the process of") takes a request as an argument and returns the identification of the process making the request. The domain of both o and p is the set of requests. See part A of figure 2.2.1.1 for examples of requests and the functions o and p .

In addition to a unique request, there is also associated with each access attempt a unique enter event and a unique exit event. The enter event corresponding to a request r is denoted by $e(r)$. Similarly the corresponding exit event is denoted by $x(r)$. Specific enter events or exit events are often denoted by e and x respectively both of which can be either primed or subscripted to denote a particular access. A variable ranging over enter events or exit events is denoted by e or x subscripted or primed. It is often convenient to talk about a null event which will be denoted by λ . The usefulness of the null event will become clear later. See part B of figure 2.2.1.1 for examples of enter and exit events.

An event sequence is a one-to-one function from the positive integers (or an initial segment of the positive integers) into the set of events. Sequences will be denoted by Greek letters. Occasionally it is useful to mention the empty sequence which will be denoted by ϵ . Note that $\epsilon \neq \lambda$. If α is a sequence, then α_i is the i^{th} member in the sequence. With some abuse of notation, $a \in \alpha$ is written if there exists an i such that $\alpha_i = a$, where a is some event. A sequence β is an initial segment of α if the domain of β is a subset of the domain of α . We write $\beta \prec \alpha$ if β is a finite initial segment

Fig. 2.2.1.1. Examples of Events and Functions on Events

Part A: Requests

$r_1, r_2[\text{write}, 1], r'[2], r_3[\text{op}, 1]$	Request events
$r, r[\text{op}], r_1[\text{read}, 4]$	Request event variables
$r_2[\text{write}, 1], r_3[\text{write}, 1]$	These requests represent two different accesses to write by the same process
$\alpha(r_2[\text{write}, 1]) = \text{write}$ $\alpha(r'[\text{read}]) = \text{read}$	Operation function
$p(r'[2]) = 2$ $p(r[\text{op}, 4]) = 4$	Process function

Part B: Enters and Exits

$e_1, e_2[\text{write}, 1]$	Enter events
$e, e'[\text{op}]$	Enter event variables
$\epsilon(r_1) = e_1$	Enter function
$x_1, x_2[\text{op}, 1]$	Exit events
$x(r[\text{op}]) = x'[\text{op}]$	Exit function

of α . Note that if α is finite, then $\alpha < \alpha$.

In the examples above, the functions α and p were defined on requests. The domains of these functions are extended in the natural way to include enter and exit events. For example $\alpha(\alpha_i) = \text{write}$, if and only if $\alpha_i = t[\text{write}, j]$ for some event type t and process id j . A new function, ι , (read "the type of") on events is also introduced. The function takes an event as an argument and returns its type. For example $\iota(r)$ request, $\iota(\epsilon(r)) = \text{enter}$, and $\iota(x[\text{op}, 1]) = \text{exit}$.

The following definitions are useful for isolating certain sets of important events in a particular sequence.

Definition 2.2.1: In a sequence α , a request r is *active* at a point α_i if there exists a $k \leq i$ with $\alpha_k = r$, but there is not a $k < i$ such that $\alpha_k = x(r)$.

Definition 2.2.2: In a sequence α , a request r is *outstanding* at a point α_i if there exists a $k \leq i$ with $\alpha_k = r$, but there is not a $k \leq i$ such that $\alpha_k = e(r)$.

Definition 2.2.3: In a sequence α , a request r is *busy* at a point α_i if there exists a $k \leq i$ with $\alpha_k = e(r)$, but there is not a $k < i$ such that $\alpha_k = x(r)$.

Note that if r is active in α at α_i , then it is either outstanding or busy at α_i .

Only some event sequences are of interest; namely, those that might arise as lifetimes of guardians. Such event sequences satisfy the following properties:

- i) Every enter event must be preceded by its corresponding request event.
- ii) Every exit event must be preceded by its corresponding enter event.
- iii) No single process ever has two requests active at the same point.

Event sequences which satisfy the above constraints are referred to as *legal guardian event sequences* or, when there is no possibility of confusion, simply as sequences. Finite legal sequences are called *histories*. For some examples of event sequences, both legal and illegal, see figure 2.2.1.2.

It will not be necessary to distinguish between two legal sequences such as these:

$r_1[w,1], r_2[w,3], e_1[w,1], x_1[w,1]$

and

$r_2[w,1], r_1[w,3], e_2[w,1], x_2[w,1]$.

This observation motivates the following definition.

Fig. 2.2.1.2. Sequence Notation and Examples of Event Sequences: Legal and Illegal

$\alpha = r_1[1], e_1[1], r_2[3], x_1[1]$	A legal event sequence.
$\alpha_1 = r_1[1]$	Examples of sequence notation applied to α .
$\alpha_3 = r_2[3]$	
$x_1[1] \in \alpha$	
$\beta = r_1[1], r_2[1], e_2[1], x_2[1]$	Illegal sequence (Violates iii).
$\xi = x_1, e_1, r_1$	Illegal sequence (Violates i & ii).
$\delta = r_1, e_1, x_1, r_1$	Not an event sequence since it is not one-to-one.

Definition 2.2.4: Two sequences α and β are *equivalent* if and only if for every natural number i :

- i) $\alpha(\alpha_i) = \alpha(\beta_i)$
- ii) $\rho(\alpha_i) = \rho(\beta_i)$
- iii) $\ell(\alpha_i) = \ell(\beta_i)$.

2.2.2 Predicates on Event Sequences

A predicate on event sequences is any function from the legal sequences to the set $\{\text{true}, \text{false}\}$. We will use the full power of second order logic to express predicates on sequences. Some useful abbreviations are introduced below. Often predicates on sequences are defined by specifying ordering constraints on the events. The most fundamental of these is "precedes". There are several different ways to define "precedes". For example, if a and b are events, then a precedes b is true of α if and only if, when both a and b are in α then a comes before b , [GRE177].¹ We, however,

1. In 3.4 it is shown that this definition of precedes can lead to predicates with undesirable characteristics.

will define it as follows: a precedes b is true of α if and only if $b \in \alpha$ implies that $a \in \alpha$ and that a comes before b . We denote this notion of precedes by \Rightarrow . More formally $a \Rightarrow b$ is true of α if and only if

$$\forall i [b \neq \alpha_i] \vee \exists i \exists j [(i < j) \wedge (a = \alpha_i) \wedge (b = \alpha_j)].$$

The abbreviation $a \Rightarrow b \Rightarrow c$ is used to mean $(a \Rightarrow b) \wedge (b \Rightarrow c)$. As an example of the use of precedes, the three properties defining legal sequences can be rewritten as follows:

- i) $r \Rightarrow c(r)$
- ii) $c(r) \Rightarrow x(r)$
- iii) $[(p(r) = p(r')) \wedge (r \Rightarrow r')] \rightarrow (x(r) \Rightarrow r').^1$

In specifying synchronization problems it is often necessary to express that two classes of requests cannot be serviced in the resource concurrently.² The mutual exclusion predicate, $Mx(r, r')$, is defined to be true of a sequence if and only if the requests r and r' are not serviced concurrently. More formally, $Mx(r, r')$ is true of α if and only if the following is true of α :

$$[(c(r) \Rightarrow c(r')) \rightarrow (x(r) \Rightarrow c(r'))] \wedge [(c(r') \Rightarrow c(r)) \rightarrow (x(r') \Rightarrow c(r))].$$

Thus the predicate $Mx(r[\text{write}], r'[\text{write}])$ means that simultaneous writes to the resource are not allowed. Often the predicate $Mx(r[\text{op1}], r'[\text{op2}])$ is written as $Mx(\text{op1}, \text{op2})$.

It is also important to be able to express that outstanding requests must be serviced according to some priority scheme. The predicate $Pr(r, r')$ is introduced for this purpose. The predicate $Pr(r, r')$ is true of α if whenever two requests r and r' are both outstanding at the same point then r will be serviced before r' . More formally $Pr(r, r')$ is true for α if and only if the following is true of α :

1. The symbol \rightarrow is used for material implication.

2. Two requests r and r' which are serviced in an event sequence are serviced concurrently in that sequence if and only if $(c(r) \Rightarrow x(r')) \wedge (c(r') \Rightarrow x(r))$.

$$[r \Rightarrow r' \Rightarrow \alpha(r) \vee r' \Rightarrow r \Rightarrow \alpha(r')] \rightarrow \alpha(r) \Rightarrow \alpha(r').$$

Often $\text{Pr}(o, o')$ is written instead of $\text{Pr}(r[o], r[o'])$.

As a brief example of problem specification, consider the readers/writers problem with reader priority. (See 4.2 for a more complete discussion of the readers/writers problem.) Briefly, there is a data base which can be accessed by either reads or writes. Concurrent reads *may* occur; concurrent writes *cannot* occur; and a read and a write *must not* occur concurrently. In addition, outstanding reads are given priority over outstanding writes. The following predicate P captures these constraints:

$$P \equiv Mx(r[\text{write}], r') \wedge \text{Pr}(\text{read}, \text{write}).$$

There are some additional useful predicates which are now defined. The predicate $\text{FIFO}(op)$ is true of a sequence if and only if all outstanding requests with the operation op are serviced in a first in first out manner; i.e.,

$$\text{FIFO}(op) \equiv r[op] \Rightarrow r'[op] \Rightarrow \alpha(r[op]) \rightarrow \alpha(r[op]) \Rightarrow \alpha(r'[op]).$$

The predicate $\text{Fr}(r)$ is used to express that requests will not be starved as long as all the requests which enter will eventually exit. More formally $\text{Fr}(r)$ is true of α if and only if the following is true of α :

$$\forall r[\alpha(r) \in \alpha \rightarrow x(r) \in \alpha] \rightarrow \forall r[r \in \alpha \rightarrow \alpha(r) \in \alpha].$$

Many of the predicates on event sequences can be put in one of the following categories according to their use in defining a synchronization problem:

- i) **Exclusion Constraints** -- Also called consistency constraints, these predicates are used to guarantee consistency of the resource by preventing certain outstanding requests from being serviced until some condition is met. An example is $Mx(r, r')$.
- ii) **Priority Constraints** -- These predicates are used to specify the order in which outstanding requests should be serviced. Examples are $\text{Pr}(o, o')$ and $\text{FIFO}(r)$.

- iii) Fairness Constraints -- These constraints are used to specify that under certain conditions, a request must eventually be serviced. An example is $Fr(r)$.

In section 3.2, optimality constraints are discussed. These constraints specify that all requests must be serviced as soon as possible. Before discussing these predicates, however, the notion of polling guardian must be introduced.

2.3 General Polling Guardian

The goal of this section is to define a framework in which to discuss all the important behavioral aspects of guardian synchronization. Such a framework will permit us to describe easily solutions to synchronization problems in 2.6. Dijkstra's Secretary [DIJK71] provides a suitable metaphor in which to discuss the framework. The Secretary is imagined as managing the access to a group of Directors. As a person comes into the waiting room to request an appointment with a Director, the Secretary either allows the requester to enter the Director's office or, if the Director is busy, has the requester wait. Presumably, as a person finishes with an appointment, he will notify the Secretary that he has exited the Director's office.

The Secretary's job can be described in terms of two tasks:

- i) Consideration of any new requests or notifications of exit. Both of which are considered one at a time on a first come first served basis. Consideration of a new request consists, first, of allowing the requester into the waiting room and, second, of modifying pertinent records. Consideration of an exit notification consists of simply updating the records which reflect the status of the Directors.
- ii) Determination of which, if any, of the waiting requesters can be allowed to proceed. One of these eligible requesters is then chosen and can have access to a Director. Records are correspondingly updated.

The Secretary alternates between these two tasks, possibly performing one of the tasks for quite some time before switching his attention to the other. The Secretary and his actions can be formalized in terms of the procedure scheme in figure 2.3.3 which is presented in an abbreviated syntax. The two tasks of the Secretary are represented by the two **repeat...until** statements. The first of these removes the first request or exit notification from the "to be considered" queue (i.e., **in**) and updates the history of the past events (i.e., α) by concatenating¹ this new request or notification onto the end of the history. We assume here that if **in** is empty then $\alpha \parallel \text{in}$ is equivalent to $\alpha \parallel \lambda$ (i.e., to α). New requests and notifications are repeatedly considered until eventually the predicate $P(\alpha)$ becomes true. Then control is passed to the second task, that of evaluating which of the waiting requests that have been considered can be allowed to continue. By examining the past history (i.e., α), the Secretary chooses a waiting requester via some strategy (i.e., G) and allows him to go on in to see a Director. Record of this action is preserved by concatenating onto the history a corresponding notification of entrance. In some cases no requester can correctly be allowed to see a director. In such a case it is assumed that $G(\alpha)$ returns λ , that $\alpha = \alpha \parallel \epsilon(\lambda)$, and that **allow**(λ) acts like a NOP. The actions of the second task are repeated until $Q(\alpha)$ is true; at which point control is passed back to the first task. The predicates P and Q are referred to as the input and output predicates respectively. The function G is called

Fig. 2.3.3. Scheme for a General Polling Guardian

```

while true do
  repeat   $\alpha \leftarrow \alpha \parallel \text{in}$     until  $P(\alpha)$ ;
  repeat  {  $r \leftarrow G(\alpha)$ ,
             $\alpha \leftarrow \alpha \parallel \epsilon(r)$ ,
            allow( $r$ ) }    until  $Q(\alpha)$ ;
end while

```

1. The symbol \parallel is used to denote concatenation.

the synchronization strategy. Initially it will be assumed that P, Q, and G can each be non-deterministic. By a non-deterministic function we mean a relation. If G is non-deterministic then $G(\alpha)$ denotes an arbitrary element of the range of G which is related by G to α [KAPU80].

When the scheme in figure 2.3.3 is seen as a resource guardian, it is important to understand how guardian events are associated with its actions. A request event is associated with the action of removing a request from the "to be considered" queue. Similarly an exit event is identified with the action of removing a notification of exit from the queue. An enter event is associated with the action of allowing a requester to continue.

The scheme given in figure 2.3.3 is the form of what will be called the general polling guardian. By giving the definitions of P, Q, and G, an actual general polling guardian can be obtained. As an example, consider the procedure M given in figure 2.3.4. Here P and Q are both defined deterministically as the constant **true**. This

Fig. 2.3.4. A Solution to $Mx(r, r')$

```

procedure M
  while true do
    repeat  $\alpha \leftarrow a_{lin}$  until true;
    repeat {  $r \leftarrow$  [ if (there is an enter event and
                        no corresponding exit event
                        in  $\alpha$ ) or (there are no out-
                        standing requests in  $\alpha$ )
                      then return  $\lambda$ 
                      else choose non-deterministically
                        some outstanding request of  $\alpha$ 
                        and return it.],
           $\alpha \leftarrow a_{lc}(r),$ 
          allow( $r$ ) } until true;
  end while
end procedure

```

means that each of the tasks will be performed once, and then control will be passed on to the other task. The function G has been given a non-deterministic definition which guarantees that no two requests will be serviced at the same time. Thus the procedure M enforces mutual exclusive access to the resource by all requesters. More formally, we could prove that

$$B(M) \subseteq Mx(r,r').^1$$

In the next section the behavior of M is examined further; and another procedure, M' , is defined which is also a solution to $Mx(r,r')$ but which has very different behavior.

2.4 Examples of General Polling Guardians

In this section the behavior of several solutions are examined and expressed as polling guardians. In particular, several solutions to the mutual exclusion problem are discussed. A monitor solution to the readers priority version of the readers/writers problem is also discussed. Besides giving examples which will familiarize the reader with general polling guardians, this section indicates the usefulness of the polling guardian in examining and comparing behaviors of resource guardians.

Consider the simple program S in figure 2.4.5 which is a critical section protected by the binary semaphore s . It is assumed that s is initialized to 1. Recall from 2.1 that a request event is associated with the instant a process starts executing the P operation. An enter event is associated with the instant a process finishes the P operation. And an exit event is associated with the instant that a process finishes decrementing the semaphore in the V operation. The behavior of S can easily be described by adapting Habermann's [HABE72] characterization of a semaphore in terms of events. The rephrasing of his central theorem results in the following:

1. Recall that $B(M)$ is the behavior of M .

Fig. 2.4.5. A Semaphore Solution to $Mx(r,r)$

procedure S

 P(s);

Critical Section

 V(s);

end procedure

A sequence $\alpha \in B(S)$ if and only if for every finite initial segment β of α which does not end immediately before an enter event, the following equality holds

$\#_{\beta} \text{enter} = \min(\#_{\beta} \text{request}, 1 + \#_{\beta} \text{exit})$, where $\#_{\beta} t$ is the number of events of type t in β .

Note that the restriction on β , "which does not end immediately before an enter event," is required because Habermann considers the P and V instructions to be indivisible relative to the theorem; i.e., the theorem is guaranteed to hold only when one of these instructions is not in progress. Since immediately prior to an enter event a process must be in the middle of executing one of these two instructions, naturally then the equation does not hold. In all other cases, however, the equation must hold. Recall the general polling guardian M of figure 2.3.4. With the above formal characterization of $B(S)$, it is straightforward but somewhat tedious to prove that $B(S) = B(M)$. Thus it is possible to describe the semantics of mutual exclusion accomplished by standard semaphores simply by giving the equivalent polling guardian. Above we have described the behavior of the procedure S when semaphores are interpreted as in [HABE72]. There is not, however, a consensus of opinion on the semantics of semaphores. Occasionally when semaphores are discussed, it is assumed that the process which has been waiting the longest is always removed first. For example, Hoare in [HOAR74] defines the semantics of monitors in terms of semaphores which he must have assumed were FIFO in behavior. Assuming s in figure 2.4.5 were such a FIFO semaphore, then S would be equivalent to M (in figure

2.3.4) if the non-deterministic choice among the outstanding requests were replaced by the purely deterministic choice of the first outstanding request.

In [HABE76] and [STAR80], the difference between strong and weak semaphores is pointed out. Since there has been some confusion in the literature between these two versions of the semaphore (see [PRES75], [KEIL76], and [HANS78]), it would be interesting to look at the general polling guardian whose behavior is identical to the behavior of S (in figure 2.4.5) when s is interpreted as a weak semaphore. In figure 2.4.6 such a polling guardian, M' , is given. Notice that the only difference between M' (in figure 2.4.6) and M (in figure 2.3.4) is the definition of the input predicate which determines when to quit the first task and to continue with the second task. In M' after an exit occurs, a request can occur before an enter occurs. In particular, the same process which has just exited can return to make a new request

Fig. 2.4.6. Weak Semaphore Solution to $Mx(r, r')$

```

procedure  $M'$ 
  while true do
    repeat  $\alpha \leftarrow \alpha \text{ in}$     until [if (last event of  $\alpha$  is
                                not an exit) then true
                                else choose non-deterministically
                                either true or false]
    repeat  $\{r \leftarrow$  [if (there is an enter event and
                        no corresponding exit event
                        in  $\alpha$ ) or (there are no out-
                        standing requests in  $\alpha$ )
                        then return  $\lambda$ 
                        else choose non-deterministically
                        some outstanding request of  $\alpha$ 
                        and return it.],
             $\alpha \leftarrow \alpha \text{ of}(r),$ 
            allow}(r)\}    until true;
  end while
end procedure

```

and possibly be chosen to enter again the data base before any other waiting process is allowed to enter. Notice that it is implicitly assumed that after an exit the input predicate will eventually, after an arbitrary but unbounded length of time, become true. In a sense the semantics of a weak semaphore makes use of a mechanism of unbounded non-determinacy [DIJK76].¹

The next example examines the behavior of a monitor solution to the readers/writers problem with readers priority. Such a solution, taken from [BLOO79], is given in figure 2.4.7.² The solution is typical of monitor and serializer solutions to the readers/writers problem in that if a number of readers are waiting when a write finishes, all the readers will be allowed into the data base before possession of the monitor is released to a new requester or to a process wishing to exit. This is reflected in the equivalent polling guardian solution given in figure 2.4.8 by the fact that control does not leave the second task until all the waiting requests that can continue have been allowed to continue.

1. In a slightly different context [KWON79] this is similar to the finite delay property of a scheduler.

2. The solution is presented in CLU syntax [LISK79].

Fig. 2.4.7. Monitor Solution to Readers Priority Version of Readers/Writers Problem

```
readers_priority = monitor is create, startread, endread, startwrite, endwrite
  rep = record{readercount:int, busy:bool,
              readers, writers:condition}

  create = proc() returns(cvt)
    return(rep${readercount:0, busy:false,
                readers, writers:condition$create()})
  end create

  startread = proc(m:cvt)
    if m.busy then condition$wait(m.readers) end
    m.readercount := m.readercount + 1
    condition$signal(m.readers)
  end startread

  endread = proc(m:cvt)
    m.readercount := m.readercount - 1
    if m.readercount = 0
      then condition$signal(m.writers)
    end %if
  end endread

  startwrite = proc(m:cvt);
    if (m.readercount > 0) | m.busy
      then condition$wait(m.writers)
    end %if
    m.busy := true
  end startwrite

  endwrite = proc(m:cvt)
    m.busy := false
    if condition$queue(m.readers)
      then condition$signal(m.readers)
    else condition$signal(m.writers)
    end %if
  end endwrite
end readers_priority
```


Fig. 2.4.8. A Solution Equivalent to the Monitor Solution

```

procedure R
  while true do
    repeat  $\alpha \leftarrow \alpha \text{llin}$  until true;
    repeat {  $r \leftarrow$  [ if (there is an outstanding read request
                        in  $\alpha$  and there are no writes
                        currently active in  $\alpha$ )
                        then return (the first outstanding read request in  $\alpha$ )
                        else if (there is an outstanding write request in  $\alpha$ 
                        and there are no currently active accesses in  $\alpha$ )
                        then return (the first outstanding write request)
                        else return  $\lambda$ ],
             $\alpha \leftarrow \alpha \text{ll}(\mathbf{r})$ ,
            allow( $\mathbf{r}$ ) } until ( $\mathbf{r} = \lambda$ );
  end while
end procedure

```

2.5 Simple Polling Guardian

In the previous section several examples of the general polling guardian which exhibited a variety of behaviors were examined. In this section some of the generality of the general polling guardian is restricted in order to obtain a simpler class of polling guardians which will be referred to as simple polling guardians. Simple polling guardians do not have as diverse a variety of behavior as general polling guardians; however, besides being simpler and easier to reason about, they are able to express most of the useful solutions to many of the synchronization problems.

A simple polling guardian is any general polling guardian which satisfies the following restrictions:

- i) The synchronization strategy must be a deterministic function of the past history of the guardian.
- ii) The input predicate must be the constant **true**.
- iii) The output predicate must also be the constant **true**.

Figure 2.5.9 gives the abbreviated syntax of a simple polling guardian. Below, each of the restrictions is discussed and motivated in turn.

Non-determinism arises naturally in discussions of synchronization because the length of arbitrary delays is often unpredictable and these independent delays can cause radically different results which are unpredictable and therefore non-deterministic in nature. In particular, the precise ordering of request and exit events for a guardian is largely non-deterministic. It is believed that non-determinism should be confined to the ordering of request and exit events and that the guardian's response to any particular ordering should be functional. By so restraining the polling guardians we greatly simplify reasoning about guardian behavior and avoid, at the local level at least, complicated issues of the semantics of non-determinism. In addition by avoiding non-determinism, it is possible in chapter 4 to define a programming construct based on the polling guardian which can be practically debugged.

Fig. 2.5.9. Scheme for Simple Polling Guardian

```
while true do
   $\alpha \leftarrow \alpha \text{llin};$ 
   $r \leftarrow G(\alpha);$ 
   $\alpha \leftarrow \alpha \text{ll}(\alpha(r));$ 
  allow(r);
end while
```

Besides being required to be deterministic, the simple polling guardian is also required to have the input predicate be the constant **true**. This requirement stems from the desire that in the absence of any other processes a synchronization mechanism should delay for as little as possible a request for service. In other words, when a new request is made, the situation should be immediately checked to see if the process can be allowed to continue with minimum delay. When a notification of exit is considered, again there is a possibility that there are requests which have been waiting and which can now be allowed to continue; consequently the mechanism should immediately start on the second task. Finally by setting the input predicate to **true**, the possibility of excluding the performance of the second task is avoided in a very simple and straightforward manner.

Now the restriction that the output predicate should also be the constant **true** is examined. The purpose of this restriction is to prevent the avoidance of the first task for arbitrary lengths of time. Returning regularly to the first task makes possible the timely consideration of new requests and exit notifications.

It is interesting that many synchronization mechanisms, notably monitors and serializers, typically define solutions which can result in the avoidance of the first task for arbitrary lengths of time. For instance, recall the monitor solution (see figure 2.4.7) to the readers/writers problem. The solution is described in terms of a general polling guardian in figure 2.4.8. Here the output predicate is not simply **true** but rather $(r = \lambda)$ where r is the result of the last evaluation of the strategy function. Thus the second task is performed until the strategy function returns a null event. A close examination reveals that in certain cases this output predicate results in undesirable consequences. For example, suppose a writer takes a relatively long time in the data base and that during this time many readers arrive. In the monitor solution, when the write finally exits from the data base, the first read request on the condition queue is awakened. This request enters the data base and awakens the next reader on the queue, and so on until all the reads have entered the data base. An awakened process on the condition queue has absolute priority over all other processes which are trying to gain control of

the monitor. Thus from the time of the awakening of the first reader on the condition queue until the last reader on the queue has entered the data base, no new requests are considered and no processes in the data base can leave it. The ignoring of new requests, some of which might be of very high priority, in order to service the requests which are currently outstanding is not good. Similarly, the ignoring of exit notifications, which results in the holding of processes in the resource, seems wrong. It is seen that solutions that can be described as having an output predicate which is not the constant **true** can result in undesirable behavior. Therefore all solutions which will be considered from now on will have an output predicate of **true**.

In figure 2.5.10 a simple polling guardian R' is given which solves the readers/writers problem with readers priority. It is interesting to compare the behavior of R' with that of the general polling guardian solution R given in figure 2.4.8. In fact it is easy to convince oneself that

$$B(R) \subseteq B(R').$$

Fig. 2.5.10. Simple Polling Guardian Solution to Readers/Writers Problem

```

procedure  $R'$ 
  while true do
     $\alpha \leftarrow \text{allin};$ 
     $r \leftarrow$  [if (there is an outstanding read request
              in  $\alpha$  and there are no writes
              currently active in  $\alpha$ )
            then return (the first outstanding read request in  $\alpha$ )
            else if (there is an outstanding write request in  $\alpha$ 
                    and there are no currently active accesses in  $\alpha$ )
            then return (the first outstanding write request)
            else return  $\lambda$ ];
     $\alpha \leftarrow \text{allc}(r);$ 
    allow( $r$ );
  end while
end procedure

```

Thus in a sense the simple guardian R' is the more general solution. The simple polling guardian provides the basis for all further discussion of resource guardians. From this point on whenever a polling guardian is mentioned, a simple polling guardian is meant unless specifically stated to the contrary.

2.6 Expression of Solutions as Simple Guardians

This section introduces a notation for describing solutions to synchronization problems. Recall from section 2.1 that a solution of a problem specification is a guardian which satisfies the specification. In section 2.4 the general polling guardian was introduced, and it was informally demonstrated how this scheme could be used to emulate the behavior of many diverse solutions. The general polling guardian is completely defined by giving descriptions of the input predicate, the output predicate, and the synchronization strategy. Thus a method for defining non-deterministic predicates and functions on histories would provide a simple solution specification language. In the previous section, however, it was argued that the simple polling guardian could provide most of the interesting solutions that might arise. Since a simple polling guardian is completely defined by the description of its functional synchronization strategy, it is sufficient to present a method for defining simple functions on histories. Below, some conventions are given for defining synchronization strategies.

Given a history α , it is often necessary to be able to talk about the outstanding requests of α and also the busy requests of α -- i.e., those which have entered the resource but have not yet exited. The set of outstanding requests of α (i.e., the waiting requests) is denoted by $W(\alpha)$, and the set of busy requests of α is denoted by $B(\alpha)$. Sometimes it is desirable to distinguish among the waiting or busy requests which will perform or are performing a particular operation. Thus the set of requests waiting to perform an operation op is denoted by $W(\alpha, op)$. Similarly $B(\alpha, op)$ denotes the set of busy requests which are performing operation op . Occasionally it is useful to define other sets which are subsets of the requests of a sequence α . These are usually denoted

by a capital letter followed by the sequence name in parenthesis. Figure 2.6.11 gives some examples.

Given a set of requests, $S(\alpha)$, it is often necessary to choose a particular element of the set based on the contents of $S(\alpha)$ and on the information implicit in the ordering defined by α . Such a choice will be called a *choice function*. A choice function takes two arguments. The first argument is a sequence α , while the second is a set of requests $S(\alpha)$ which appear in α . A choice function always returns either a request which is a member of its second argument or λ , the null event. Thus if the second argument is the empty set, a choice function must return λ .

A useful choice function is **min** which returns the first request of its second argument. In other words for every sequence α and set of requests $S(\alpha)$,

$$\text{min}(\alpha, S(\alpha)) = r \text{ where } r \in S(\alpha) \text{ and if } r' \in S(\alpha), r \neq r' \text{ then } r \Rightarrow r'.$$

Since the first argument of a choice function is usually obvious from context, it is often omitted. Thus $\text{min}(S(\alpha))$ is written instead of $\text{min}(\alpha, S(\alpha))$.

Fig. 2.6.11. Subsets of Requests

$$\alpha = r_1[\text{write}, 2], e_1[\text{write}, 2], r_2[\text{read}, 1], r_3[\text{write}, 6]$$

$$U(\alpha) = W(\alpha) \cup B(\alpha)$$

$$S(\alpha) = \{r \in \alpha \mid p(r) = 1\}$$

$$W(\alpha) = \{r_2, r_3\}$$

$$W(\alpha, \text{write}) = \{r_3\}$$

$$B(\alpha) = \{r_1\}$$

$$B(\alpha, \text{write}) = \{r_1\}$$

$$B(\alpha, \text{read}) = \{\}$$

$$U(\alpha) = \{r_2, r_3, r_1\}$$

$$S(\alpha) = \{r_2\}$$

To assist in defining choice functions the **if...then...elseif** function is introduced. It is evaluated in a manner similar to the evaluation of the **COND** construct of LISP. An example will make this clear. Consider the choice function c on $S(\alpha)$ defined using two other choice functions c_1 and c_2 , and the predicates P and Q :

$$c(S(\alpha)) = \begin{array}{ll} \text{if } P(S(\alpha)) \text{ then } c_1(S(\alpha)) \\ \text{elseif } Q(S(\alpha)) \text{ then } c_2(S(\alpha)) \\ \text{otherwise } \lambda. \end{array}$$

Now $c(S(\alpha))$ is $c_1(S(\alpha))$ whenever $P(S(\alpha))$ is true, and is $c_2(S(\alpha))$ if $P(S(\alpha))$ is false but $Q(S(\alpha))$ is true. If both $P(S(\alpha))$ and $Q(S(\alpha))$ are false, then $c(S(\alpha))$ is λ , the null event.

A synchronization strategy G applied to α always returns a member of $W(\alpha)$, the set of outstanding requests of α . Thus a synchronization strategy can be thought of as a choice function where, when the first argument is α , the second argument is always $W(\alpha)$. Therefore the methods for defining choice functions are also applicable for defining synchronization strategies. Figure 2.6.12 is the definition of a synchronization strategy which defines a solution to the readers/writers problem with readers priority. This is just a rewriting of the solution, R' , given in figure 2.5.10.

This section has presented a brief introduction to a way in which synchronization strategies might be described. Although we will not do so, it is possible to extend the techniques presented here to include more involved methods (e.g., recursion) for defining synchronization strategies.

Fig. 2.6.12. Specification of a Solution to Readers/Writers Problem

$$G(\alpha) = \begin{array}{ll} \text{if} & (W(\alpha, \text{read}) \neq \emptyset) \wedge (B(\alpha, \text{write}) = \emptyset) \text{ then } \min(W(\alpha, \text{read})) \\ \text{elseif} & (B(\alpha) = \emptyset) \text{ then } \min(W(\alpha, \text{write})) \\ \text{otherwise} & \lambda. \end{array}$$

3. Guardian/Environment Game

This chapter gives a formal definition of guardian and many of the other terms which were introduced informally in 2.1. The formalization presented here stems from viewing guardian interaction with its environment as a two person game. A guardian is defined formally as a functional strategy for the second player. Once a guardian is defined formally, it is possible to address the issue of concurrency in a problem specification. The notion of optimal solution is defined in order to capture the intuition of what constitutes a good solution. Several examples are given showing how concurrency can be specified by requiring that the solutions to a problem be optimal. In the third section a certain subset of well-behaved problem specifications is defined as continuous. A theorem is then proved which characterizes the solutions to all continuous specifications. In the last section simple predicates are defined. Every simple predicate is continuous, and it is easy to determine whether or not a solution is optimal for simple predicates.

3.1 Definitions

The interaction of a resource guardian and the rest of the system -- i.e., the guardian's environment -- can be viewed as a game between two players which will be identified as players I and II. Player I is associated with the environment while player II is associated with the guardian. The two players alternate moves building a legal event sequence. Player I moves by concatenating either a request or exit event onto the sequence being built while player II plays only enter events. On each move, the player moving has complete knowledge of the sequence constructed thus far. A predicate P determines the winner of the game. Thus a different game arises from each problem specification P -- i.e., from each predicate on event sequences. Player II wins an instance of a game defined by a predicate P if the sequence built by the game satisfies P . If the sequence built does not satisfy P , then player I wins.

Play starts with player I concatenating an event onto the empty sequence. A legal move by player I is any request or exit event which when concatenated onto the game sequence -- i.e., the sequence being built -- results in a legal event sequence. A legal move by player II is any enter event which when concatenated onto the game sequence results in a legal event sequence. If either player cannot make a move, he must pass. This is accomplished by playing the null event. Note that a player can also pass even when he has a legal move. When both players pass on consecutive moves, this particular instance of the game is over and the sequence built is finite. Otherwise the game continues forever and an infinite sequence is built. Figure 3.1.1 gives an example of an instance of the game defined by the predicate $Mx(o,o)$. Player I wins this instance since the sequence α does not satisfy the predicate.

A strategy for player I, the environment, is any function E from H (the set of histories) into the set of non-enter type events such that if $\alpha \in H$ then $\alpha \parallel E(\alpha) \in H$. A strategy for player II, the guardian, is any function G from H into the set of enter events along with the null event such that if $\alpha \in H$ then $\alpha \parallel G(\alpha) \in H$. Now a formal definition of a simple guardian can be given:

Fig. 3.1.1. An Instance of Guardian/Environment Game Defined by $P \equiv Mx(o,o)$

	Player I	Player II
1.	$r_1[o]$	$e_1[o]$
2.	$r_2[o]$	λ
3.	$r_3[o]$	$e_3[o]$
4.	λ	$e_2[o]$
5.	$x_1[o]$	λ
6.	$x_2[o]$	λ
7.	$x_3[o]$	λ
8.	λ	

$\alpha = r_1[o], e_1[o], r_2[o], r_3[o], e_3[o], e_2[o], x_1[o], x_2[o], x_3[o]$

Definition 3.1.1: A *simple guardian* is any functional strategy for player II.

If player I adopts a strategy E and player II adopts a strategy G , the sequence built in playing the game is completely determined and is denoted by $[E, G]$. Note that either $[E, G]$ is infinite or $G([E, G]) = \lambda$ and $E([E, G]) = \lambda$.

Definition 3.1.2: A *lifetime* of a simple guardian G is any sequence α for which there exists an environment strategy E such that $\alpha = [E, G]$.

Definition 3.1.3: The *behavior* of a guardian G , denoted by $B(G)$, is the set of all lifetimes of G .

Definition 3.1.4: A guardian strategy G is a *winning* strategy for the game defined by a predicate P if and only if for every environment strategy E , $P([E, G])$ is always true.

Definition 3.1.5: When G is a winning guardian strategy for a game P , then it is said that G is a *solution* to the problem specification P .

To summarize, a game has been described which views a resource guardian as trying to enforce a particular type of behavior while the possibly malicious environment tries to defeat the guardian's efforts. From this definition of a game, formal definitions of simple guardian, lifetime, behavior, and solution have emerged. Below we establish the connection between the simple guardian as defined above and the simple polling guardian as defined in 2.5. Let G' be a functional strategy for player II. Recall the scheme given for the simple polling guardian in figure 2.5.9. Consider the simple polling guardian which results from using G' as the synchronization strategy. Now the behavior, as described in 2.1, of this simple polling guardian is the identical set of sequences as $B(G')$ defined by 3.1.3. This is true because the scheme for simple polling guardian enforces the alternation of moves by the environment and the functional strategy. Thus a legal game sequence will always result from the simple polling guardian.

It is often useful to speak of the domain and results of a strategy for player II. Informally, the domain of a strategy G for player II is the set of sequences that might arise in which it is player II's move, and the results of a strategy G are the set of sequences that can arise after a move by player II. More formally:

Definition 3.1.6: The *domain* of G , denoted by $\text{dom}(G)$, is defined inductively below.

- i) $E(\epsilon) \in \text{dom}(G)$, for all strategies, E , for player I.
- ii) $\alpha \in \text{dom}(G) \rightarrow \alpha \parallel G(\alpha) \parallel E(\alpha \parallel G(\alpha)) \in \text{dom}(G)$,
where $\alpha = \alpha' \parallel E(\alpha')$, for all strategies, E , for player I.

Definition 3.1.7: The *results* of a strategy G for a player II, denoted by $\text{res}(G)$, is defined as follows:

$$\alpha \in \text{res}(G) \text{ if and only if } \exists \beta, \beta \in \text{dom}(G) \wedge \alpha = \beta \parallel G(\beta).$$

Occasionally it is useful to record *all* the moves of each player, including the passes. This results in an *expanded* game sequence. Notice that there is no new information contained in an expanded sequence. This is because the players alternate moves and because they both follow functional strategies. Because they follow functional strategies, two null events cannot appear in a row except at the end of the sequence. There is a one-to-one correspondence between the game sequences and the expanded game sequences. See figure 3.1.2 for examples. Also note that null events played by player I are differentiated from those played by player II. Again this adds no new information. The usefulness of the expanded representation of a sequence will be seen in the next section.

3.2 Specification of Concurrency

This section returns to the issue of problem specification. In particular, attention is focused on how to specify that a certain amount of concurrency is to be required in a solution. The discussion is centered around an example; namely, the readers/writers problem with readers priority which was introduced in Chapter 2.

Fig. 3.1.2. Sequences and Their Corresponding Expanded Versions

$$\alpha = r_1[o], e_1[o], r_2[o], r_3[o], e_3[o], e_2[o], x_1[o], x_2[o], x_3[o]$$

$$\alpha' = r_1[o], e_1[o], r_2[o], \lambda_{11}, r_3[o], e_3[o], \lambda_1, e_2[o], x_1[o], \lambda_{11}, x_2[o], \lambda_{11}, x_3[o], \lambda_{11}, \lambda_1$$

$$\beta = r_1, e_1, r_2, x_1, e_2, x_2$$

$$\beta' = r_1, e_1, r_2, \lambda_{11}, x_1, e_2, x_2, \lambda_{11}, \lambda_1$$

$$\xi = r_1, e_1, r_2, r_3, e_2, e_3, x_1, r_4, x_2, x_3, e_4, x_4$$

$$\xi' = r_1, e_1, r_2, \lambda_{11}, r_3, e_2, \lambda_1, e_3, x_1, \lambda_{11}, r_4, \lambda_{11}, x_2, \lambda_{11}, x_3, e_4, x_4, \lambda_{11}, \lambda_1$$

Recall that the following predicate defines the readers priority version of the readers/writers problem:

$$R \equiv Mx(r[\text{write}], r') \wedge \text{Pr}(\text{read}, \text{write}).$$

Notice that a solution which allows only one process at a time satisfies the specification. Thus the guardian O , specified in figure 3.2.3, is a solution to R . It would be useful to be able to modify R so as to rule out solutions such as this one which excludes the possibility of concurrent reads.¹

Fig. 3.2.3. One at a Time Solution to Readers/Writers Problem

$$O(\alpha) \leftarrow \begin{cases} \text{[if } (B(\alpha) = \emptyset) \wedge (W(\alpha, \text{read}) \neq \emptyset) \text{ then } \min(W(\alpha, \text{read})) \\ \text{elseif } B(\alpha) = \emptyset \text{ then } \min(W(\alpha)) \\ \text{otherwise } \lambda] \end{cases}$$

1. Another pathological solution, O' , is the one which simply prevents writes but allows all reads to proceed unimpeded.

First recall from 2.2.2 what it means for two requests r and r' to be serviced concurrently: If $e(r)$ and $e(r')$ are both in a sequence α , then

$$e(r) \Rightarrow x(r') \wedge e(r') \Rightarrow x(r).$$

With this in mind, one approach to specifying concurrency would be to say that whenever two read requests are both outstanding at the same time, they should be serviced concurrently; i.e.,

$$Q \equiv [r[\text{read}] \Rightarrow r'[\text{read}] \Rightarrow e(r) \vee r'[\text{read}] \Rightarrow r[\text{read}] \Rightarrow e(r')] \rightarrow \\ e(r) \Rightarrow x(r') \wedge e(r') \Rightarrow x(r).$$

Now the predicate R can be modified to obtain $R \wedge Q$. Note that the guardian O defined in figure 3.2.3 is not a solution for $R \wedge Q$.¹ Thus it would seem that this approach solves the problem. In fact this is essentially the approach to specifying concurrency taken by Hewitt and Atkinson [HEWI79a]. In their paper they go on to prove that a serializer can easily be used to guarantee concurrency because processes which are in the "waiting room" are given absolute priority for gaining access to the serializer over the processes trying to access the serializer from any other locations.²

As was seen in section 2.5, this approach can lead to the undesirable situation of ignoring new requests and new notifications of exit. In fact it is this ignoring of new exits until all the reads have entered that makes it possible for a serializer to satisfy $R \wedge Q$. If exits were allowed to occur freely, the first read on the condition queue might enter and exit before the last read were able to enter. This would violate Q . Thus in order to satisfy Q , a solution must prevent exits until all the outstanding reads are allowed to enter. Surely specifying that a solution must be able to serve reads concurrently should not imply that the solution must also at some time prevent exits from occurring. The above approach to specifying concurrency is therefore discarded.

1. The solution O' described in the previous footnote is not, however, ruled out by this new predicate.

2. A similar claim can be made about monitors [HOAR74] since processes waiting on condition queues are given absolute priority over others which are trying to gain control of the monitor (see 2.4, 2.5).

In order to develop another method for specifying concurrency, we return to the discussion of winning strategies for guardians.

A winning strategy is greedy if it passes only when it must in order to win. More formally:

Definition 3.2.1: A winning strategy G is *greedy* if and only if for every $\alpha \in \text{dom}(G)$ such that $G(\alpha) = \lambda$ there exists no other winning strategy G' with $\alpha \in \text{dom}(G')$ so that $G'(\alpha) \neq \lambda$.

A winning strategy is lazy if it always passes unless it must not in order to win. More formally:

Definition 3.2.2: A winning strategy G is *lazy* if and only if for every $\alpha \in \text{dom}(G)$ such that $G(\alpha) \neq \lambda$ there exists no other winning strategy G' with $\alpha \in \text{dom}(G')$ so that $G'(\alpha) = \lambda$.

The existence of a winning strategy does not guarantee the existence of a lazy strategy (e.g., $\text{Fr}(r)$).¹ To see that this example is correct, note that if a lazy strategy existed, it would put off servicing a request for as long as possible. Since on any particular move, a lazy strategy would not need to service any request, it would procrastinate forever and would not be fair as required. Similarly, the existence of a winning strategy does not guarantee the existence of a greedy strategy (e.g., $\neg \text{Fr}(r)$).

Consider for a moment, as examples of greedy and lazy strategies, the two solutions M and M' to $Mx(r, r')$ which are specified in figure 3.2.4. The solution M is greedy while the solution M' is lazy. In most cases when a person writes the specification of a problem, he has in mind only the greedy solutions. One way to

1. Recall from 2.2.2 that $\text{Fr}(r)$ states that if every enter is eventually followed by a corresponding exit, then every request will eventually be followed by a corresponding enter.

Fig. 3.2.4. Greedy and Lazy Solutions to $Mx(r,r')$

$$M(\alpha) = \begin{array}{ll} \text{if } B(\alpha) = \emptyset \text{ then } \min(W(\alpha)) \\ \text{otherwise } \lambda \end{array}$$

$$M'(\alpha) = \lambda$$

eliminate the lazy solutions is to define a predicate Q such that if G is a winning solution for $P \wedge Q$, then G is a greedy winning solution for P . This is precisely what needs to be done to eliminate the pathological solutions to the readers/writers problem. This observation inspires the following definition.

Definition 3.2.3: Given a predicate P , the predicate Q is the *optimality constraint* for P if the set of solutions of $P \wedge Q$ is the set of greedy winning strategies for P .

Note that greedy winning strategies allow as much concurrency as possible. Thus given a predicate P , the conjunction of P and the optimality constraint for P can be viewed as specifying that as much concurrency as possible be present in all solutions.

It turns out that if one uses the expanded representation of the sequences, it is easy to express the optimality constraint for R , the specification of the readers/writers problem discussed above. What is required is that whenever the guardian passes there must either be a write that is currently busy in which case no new requests can be allowed, or there must be no outstanding requests at all; i.e.,

$$Q \equiv \forall \lambda_1 \{ \exists r [\text{write}][\alpha(r) \Rightarrow \lambda_1 \Rightarrow x(r)] \vee \\ \forall r [r \Rightarrow \lambda_1 \rightarrow \alpha(r) \Rightarrow \lambda_1] \}.$$

We say that a sequence satisfies such a predicate if its expanded representation satisfies the predicate. The predicate $P \wedge Q$ is now an accurate specification of what is usually meant by the readers/writers problem with readers priority. Notice that not only has the solution O given in figure 3.2.3 been eliminated; so has the solution O' alluded to in the footnote. The guardian G described in figure 2.6.12 in the previous chapter is,

however, a solution.

It is interesting to look at the optimality constraint for mutual exclusion as well. Here the guardian must pass only if there is either a request which is busy or no request waiting. In other words:

$$Q' \equiv \forall \lambda_1 \{ \exists r [c(r) \Rightarrow \lambda_1 \Rightarrow x(r)] \vee \\ \forall r [r \Rightarrow \lambda_1 \rightarrow c(r) \Rightarrow \lambda_1] \}$$

In this section two approaches for differentiating between proper and pathological solutions have been discussed. This lead to a discussion of maximally concurrent solutions, which our intuition tells us are the best (i.e., optimal) solutions. In an attempt to capture the notion of a maximally concurrent solution the following definition is given:

Definition 3.2.1: A guardian G is an *optimal solution* to the problem specification P if and only if G is a greedy winning strategy for player II in the game defined by P .

3.3 Continuity of Specifications

In the previous sections several varieties of problem specifications were discussed. In this section continuity of a specification is defined. Several examples to illustrate continuity are given. Then a theorem which characterizes the solutions to a continuous predicate is proved. This theorem forms the basis for many of the correctness proofs in Chapter 4.

Definition 3.3.1: A problem specification P (i.e., a predicate on sequences) is *continuous* precisely when $P(\epsilon)$ is true and when a sequence α satisfies P if and only if all finite initial segments of α satisfy P .

Most of the specifications dealt with in this paper, with a few important exceptions (e.g., $Fr(r)$), are continuous as the following theorems ascertain.

Theorem 3.3.1: If the specifications P and Q are both continuous, then $P \wedge Q$ is continuous.

Theorem 3.3.2: The following predicates are continuous:

- i) $a \Rightarrow b$
- ii) $Mx(r, r')$
- iii) $Pr(r, r')$
- iv) $FIFO(r)$

The proofs of these theorems are straightforward and of no special interest; therefore, they are omitted.

The property of continuity is very powerful. Besides encompassing a large number of practically useful specifications, it also gives rise to the following theorem which will prove helpful in verifying implementations.

Theorem 3.3.3: If P is a continuous predicate and G is a guardian, then the following two statements are true:

- i) $(\alpha \in \text{dom}(G) \wedge P(\alpha)) \rightarrow P(\alpha \parallel G(\alpha))$
- ii) $(\alpha \in \text{res}(G) \wedge P(\alpha)) \rightarrow P(\alpha \parallel E(\alpha))$ for any environment strategy E

if and only if

G is a solution for P .

This theorem rule looks more complicated than it is. It says that if P is continuous, one can always tell whether G , a guardian, is a solution for P simply by verifying the following statement:

If P is true of a sequence α , then for any environment strategy E , the sequence resulting after the next move (be it either the guardian's move or environment's move) will satisfy P .

Below we give a proof of theorem 3.3.3.

Proof: First assume that G is a solution for P . Denote by α a sequence such that $\alpha \in \text{dom}(G)$ and $P(\alpha)$. Now there exists a sequence $\beta \in B(G)$ with $(\alpha \parallel G(\alpha)) \prec \beta$.¹ Since G is a solution to P , we know that $P(\beta)$ holds; but P is continuous; hence, $P(\alpha \parallel G(\alpha))$ is true. Thus i) holds. That ii) holds may be shown in a similar manner.

Now we assuming that both i) and ii) hold, we show that G is a solution for P . We proceed by contradiction supposing that G is not a solution -- i.e., that there exists a sequence β in $B(G)$ such that $P(\beta)$ is false. Since P is continuous, there must be some finite initial segment of β for which P also does not hold. Let α be a shortest such segment. Now α is not ϵ since $P(\epsilon)$; so there is a non-null event, a , such that $\alpha = \alpha' \parallel a$. From the way that α has been chosen, we know that $P(\alpha')$ is true. If a is a request or exit event, then $\alpha' \in \text{res}(G)$ and ii) is contradicted. But if a is an enter event, then $\alpha' \in \text{dom}(G)$ and i) is contradicted. Therefore G must be a solution for P .

There are important predicates which are not continuous but which have solutions. For such predicates the theorem is not applicable. The most striking example is $\text{Fr}(r)$. For instance

$$\alpha = r_1, r_2, \epsilon(r_1), x(r_1), r_3, \epsilon(r_2), x(r_2), r_4, \dots, r_i, \epsilon(r_{i-1}), x(r_{i-1}), \dots$$

satisfies $\text{Fr}(r)$ but $\text{Fr}(r)$ is not true for all of the finite initial segments of α . The guardian

$$G(\alpha) = \text{if } W(\alpha) \neq \emptyset \text{ then } \min(W(\alpha))$$

is, however, a solution for $\text{Fr}(r)$.

The following lemma will be useful in the next chapter.

Lemma 3.3.4: If G is a solution to a continuous predicate P and $\alpha \in \text{dom}(G)$, then $P(\alpha)$ is true.

1. Recall from 2.2.1 that $\beta \prec \alpha$ if and only if β is a finite initial segment of α .

Proof: Suppose, by way of contradiction, that $\alpha \in \text{dom}(G)$ but $\neg P(\alpha)$. Now since P is continuous, there must be a finite initial segment of α for which P does not hold. Let α' be a shortest such initial segment. Since P is continuous, $P(\epsilon)$ must hold. Thus there must exist an event $a \neq \lambda$ so that $\beta \parallel a = \alpha'$. Now since G is a solution to P and P is continuous, we know from theorem 3.3.3 that for every α and for every environment E

$$\text{i) } (\alpha \in \text{dom}(G) \wedge P(\alpha)) \rightarrow P(\alpha \parallel G(\alpha))$$

$$\text{ii) } (\alpha \in \text{res}(G) \wedge P(\alpha)) \rightarrow P(\alpha \parallel E(\alpha)).$$

Now if a is an enter event then $\alpha' = \beta \parallel G(\beta)$ and i) above is violated. Similarly if a is a request or exit, ii) is violated. Therefore we have a contradiction. We conclude that $\alpha \in \text{dom}(G)$ implies $P(\alpha)$.

3.4 Simple Predicates

In this section a subset of the continuous predicates (i.e., the simple predicates) is defined. First it is shown that all of the continuous predicates discussed so far are simple. Then two theorems are presented. The first characterizes solutions to simple predicates, and the second characterizes optimal solutions to simple predicates. Another theorem states that every simple predicate has an optimal solution.

Definition 3.4.1: A predicate P is *simple* if and only if

- i) P is continuous, and
- ii) For every event a , $P(\alpha) \wedge (t(a) \neq \text{enter}) \rightarrow P(\alpha \parallel a)$.

The next two theorems follow directly from the definition of simple.

Theorem 3.4.1: If P and Q are both simple then $P \wedge Q$ is simple.

Theorem 3.4.2: The following predicates are simple:

- i) $a \Rightarrow b$
- ii) $Mx(r, r')$
- iii) $Pr(r, r')$
- iv) $\text{FIFO}(r)$

An interesting example of a predicate which is continuous but not simple arises from an alternate definition of precedes which was mentioned previously (see 2.2.2). Let \ll denote the alternate form for precedes which is defined as follows: For two events a and b , $a \ll b$ is true of α if and only if, when a and b are both in α then a comes before b . Now we can define an alternate version of the mutual exclusion predicate based on the above precedes as follows:

$$Mx'(r,r') \equiv [(c(r) \ll c(r')) \rightarrow (x(r) \ll c(r'))] \wedge [(c(r') \ll c(r)) \rightarrow (x(r') \ll c(r))].$$

The predicate $Mx'(r,r')$ can be shown to be continuous; however, the following example shows that it is not simple. Consider the sequence

$$\alpha = r, r', c(r), c(r')$$

Now $Mx'(r,r')$ is true of α but is not true of $\alpha \parallel x(r)$.

The following theorem characterizes the solutions to simple predicates. It is actually just a corollary to Theorem 3.3.3.

Theorem 3.4.3: If P is a simple predicate and G is a guardian, then

$$(\alpha \in \text{dom}(G) \wedge P(\alpha)) \rightarrow P(\alpha \parallel G(\alpha))$$

if and only if

G is a solution for P .

The next theorem provides more useful information about solutions to simple predicates. Recall from 2.2.1 that $\beta \prec \alpha$ means that β is a finite initial segment of α .

Theorem 3.4.4: If P is a simple predicate such that $P(\alpha) \equiv \forall \beta [\beta \prec \alpha \rightarrow Q(\beta)]$ and G is a guardian, then

$$(\alpha \in \text{dom}(G) \wedge Q(\alpha)) \rightarrow Q(\alpha \parallel G(\alpha))$$

if and only if

G is a solution for P .

Proof: First we assume that for all α , $\alpha \in \text{dom}(G) \wedge Q(\alpha) \rightarrow Q(\alpha \parallel G(\alpha))$. To prove that G is a solution to P , we first prove that $\alpha \in \text{dom}(G) \wedge P(\alpha) \rightarrow P(\alpha \parallel G(\alpha))$, then we apply theorem 3.4.3. Assume $\alpha \in \text{dom}(G) \wedge P(\alpha)$. Now since $P(\alpha) \equiv \forall \beta [\beta \prec \alpha \rightarrow Q(\beta)]$, we know that $\alpha \in \text{dom}(G) \wedge P(\alpha) \wedge Q(\alpha)$ is true. But $\alpha \in \text{dom}(G) \wedge Q(\alpha) \rightarrow Q(\alpha \parallel G(\alpha))$. Therefore $P(\alpha) \wedge Q(\alpha \parallel G(\alpha))$ is true. Again from $P(\alpha) \equiv \forall \beta [\beta \prec \alpha \rightarrow Q(\beta)]$, we can conclude that $P(\alpha \parallel G(\alpha))$ is true. Thus $\alpha \in \text{dom}(G) \wedge P(\alpha) \rightarrow P(\alpha \parallel G(\alpha))$ is true; and from theorem 3.4.3, G is a solution to P .

Next we assume that G is a solution to P and that $\alpha \in \text{dom}(G) \wedge Q(\alpha)$. We must show that $Q(\alpha \parallel G(\alpha))$. Now by lemma 3.3.4 $P(\alpha)$ is true. Since G is a solution to P , we can use theorem 3.4.3 to conclude that $P(\alpha \parallel G(\alpha))$ is true, which means that $Q(\alpha \parallel G(\alpha))$ is true.

Note that in the above theorem, Q does not have to be simple or even continuous. Also note that Q can be substantially weaker than P . In general we are interested in the weakest such Q . As will be seen in 4.6, this theorem is quite useful in the verification of solutions. The next theorem characterizes the optimal solutions of simple predicates.

Theorem 3.4.5: If G is a solution to a simple predicate P , then for all $\alpha \in \text{dom}(G)$

$$(G(\alpha) = \lambda) \equiv \forall r [r \in W(\alpha) \rightarrow \neg P(\alpha \parallel c(r))]$$

if and only if

G is optimal.

Proof: Assume G is optimal. Since G is a solution to P , a simple predicate, we know that $\forall r [r \in W(\alpha) \rightarrow \neg P(\alpha \parallel c(r))] \rightarrow (G(\alpha) = \lambda)$ from theorem 3.4.3. Now, by way of contradiction, suppose there is a $\beta \in \text{dom}(G)$ such that $G(\beta) \neq \lambda$ but there is an $r \in W(\beta)$ with $P(\alpha \parallel c(r))$ holding. Define the strategy G' as follows:

$$G'(\alpha) = \begin{cases} G(\alpha) & \text{if } \alpha \neq \beta \\ c(r) & \text{if } \alpha = \beta. \end{cases}$$

Now since G is a solution of P , theorem 3.4.3 and the definition of G' imply that for any $\alpha \in \text{dom}(G')$, $P(\alpha) \rightarrow P(\alpha \parallel G'(\alpha))$. Thus by theorem 3.4.3 G' is a solution to P . This implies G is not optimal which is a contradiction. Therefore for all $\alpha \in \text{dom}(G)$

$$(G(\alpha) = \lambda) \equiv \forall r [r \in W(\alpha) \rightarrow \neg P(\alpha \parallel c(r))].$$

Next assume that for every $\alpha \in \text{dom}(G)$

$$(G(\alpha) = \lambda) \equiv \forall r[r \in W(\alpha) \rightarrow \neg P(\alpha \parallel c(r))].$$

Also assume that $\alpha \in \text{dom}(G)$ and $G(\alpha) = \lambda$. Let G' be any other solution to P with $\alpha \in \text{dom}(G')$. Since $\alpha \in \text{dom}(G)$, and since G is a solution to P , a simple predicate, we know from lemma 3.3.4 that $P(\alpha)$ is true. But G' is also a solution to P . So by theorem 3.4.3, if $G'(\alpha) = c(r) \neq \lambda$, then $P(\alpha \parallel c(r))$ holds which contradicts $(G(\alpha) = \lambda) \equiv \forall r[r \in W(\alpha) \rightarrow \neg P(\alpha \parallel c(r))]$. Thus $G'(\alpha)$ must be λ . Therefore, from the definition of greedy, G must be optimal.

Often it is useful to know whether or not a predicate has a solution and, if it does have a solution, whether or not it has an optimal solution. The following theorem addresses this issue for simple predicates.

Theorem 3.4.6: If P is a simple predicate, then P has an optimal solution.

Proof: We will prove the theorem by defining an optimal solution for P . Given a history α , define $S(\alpha)$, a subset of $W(\alpha)$, as follows:

$$S(\alpha) = \{r \in W(\alpha) \mid P(\alpha \parallel c(r))\}.$$

Define the strategy G as follows:

$$G(\alpha) = \begin{cases} \text{if } S(\alpha) \neq \emptyset \text{ then } \min(S(\alpha)) \\ \text{otherwise } \lambda. \end{cases}$$

From the definition of G , we know that if $P(\alpha)$ then $P(\alpha \parallel G(\alpha))$. Thus by theorem 3.4.3, G is a solution to P . Also from the definitions of G and $S(\alpha)$, we know that

$$(G(\alpha) = \lambda) \rightarrow \forall r[r \in W(\alpha) \rightarrow \neg P(\alpha \parallel c(r))].$$

Since G is a solution to P , this means that

$$(G(\alpha) = \lambda) \equiv \forall r[r \in W(\alpha) \rightarrow \neg P(\alpha \parallel c(r))].$$

Therefore from theorem 3.4.5 this means G is optimal.

Note that if a simple predicate P is decidable, this optimal solution will be computable; i.e., there will exist a program which can actually implement the functional strategy. To see that this is so, consider that the solution need only proceed as follows: For a given α , test $P(\alpha \parallel c(r))$ for each $r \in W(\alpha)$. If for some $r \in W(\alpha)$, $P(\alpha \parallel c(r))$ is true, then return r ; otherwise, if for all $r \in W(\alpha)$, $\neg P(\alpha \parallel c(r))$, then return λ .

4. A Synchronization Mechanism

In this chapter the simple polling guardian developed previously is used as the semantic basis for a synchronization mechanism for coordinating the execution of procedures in a multi-processing environment. A procedure is said to be protected by the mechanism if every process which invokes the procedure must be granted permission by the mechanism before it can execute the procedure. The invocation of a protected procedure by a process results in the process being suspended and causes the creation of a request signal which is sent to the mechanism. The return of a process from a protected procedure causes an exit signal to be sent to the mechanism. A process dedicated to the mechanism examines the signals sent to it and decides which of the suspended invocations, if any, may proceed. It is this process which actually implements the synchronization strategy used for coordinating the protected procedures. If a process can proceed, it is activated and allowed to enter the procedure.

The first section of this chapter describes the synchronization mechanism in further detail and includes a complete description of both the semantics and a possible syntax. A simple example is given and some implementation issues are also mentioned. The next four sections provide more detailed examples of the construct. The synchronization problems discussed include the readers/writers problem, the dining philosophers problem, and the disk scheduler problem. The next to the last section presents an approach for verifying the correctness of implementations. Several interesting examples are examined. The last section evaluates the construct and provides some concluding remarks.

4.1 The Synchronization Mechanism

This section defines a synchronization mechanism which is based on the simple polling guardian described in 2.5. First an overview is given. Then a more detailed description and a simple example are presented. The last subsection discusses briefly some of the implementation issues.

4.1.1 Overview

The synchronization mechanism defined here is based on the description of the simple polling guardian presented in 2.5. The mechanism will be defined in such a way as to make possible the coordination of the execution of procedures. To understand how a polling guardian might be used in order to accomplish synchronization, consider the following scenario.

Suppose that it is necessary to limit execution of a procedure named *critical_section* so that only one process at a time can be in *critical_section*. This can be done as follows. First create a procedure like the one presented in figure 4.1.1.1, and then dedicate a process to executing this procedure.¹ Next all invocations to *critical_section* are modified. The invocations are changed so that instead of simply calling *critical_section*, a process will first create a message or signal, stating that it wishes to execute *critical_section*. Next the process puts this message² on the queue in of the procedure protector.³ Then the process deactivates itself. Immediately after the deactivate command is a jump to *critical_section*. At the return point from *critical_section*, code is inserted so that the process will build another message. This message states that the process has finished executing *critical_section*. The process puts it on the queue in of protector which will interpret it as an exit. Figure 4.1.1.2

1. Note that this procedure has the same form as the simple polling guardian given in figure 2.5.9.

2. The message will be interpreted by protector as a request.

3. Operations on the queue are considered to be atomic (see 4.1.3).

Fig. 4.1.1.1. Simple Pol' : Guardian Procedure Solving $M_X(r,r)$

```

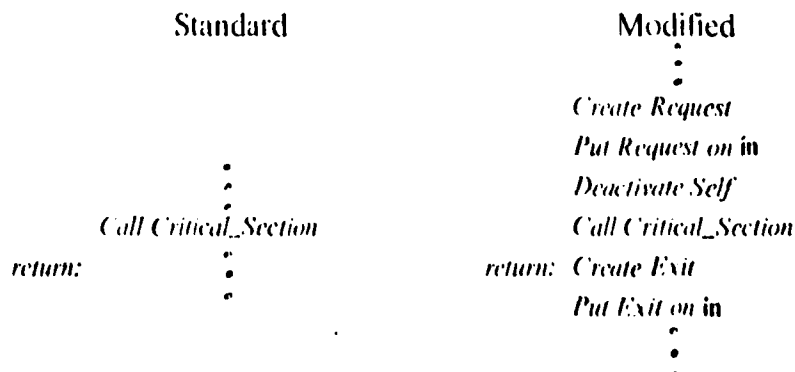
procedure  $\text{prohibit}$ 
 $\alpha \leftarrow \epsilon$ 
while true do
     $\alpha \leftarrow \alpha \text{in};$ 
     $r \leftarrow$  if  $B(\alpha) = \emptyset$  then  $\min(W(\alpha))$ 
    otherwise  $\lambda$ 
     $\alpha \leftarrow \alpha \text{ll}(r);$ 
     $\text{allow}(r);$ 
end while

```

summarizes the differences between an invocation of `critical_section` before modification and after modification.

From the above it can be seen that when a process goes to invoke `critical_section`, it first passes a request message to protector and deactivates itself. The protector takes the request off `in` and concatenates it to the history of past messages. Eventually, when there are no processes busy in `critical_section`, the request will be chosen by the synchronization strategy. The history is updated to reflect this change and `allow` is executed. The execution of `allow` reactivates the sleeping process. On being awakened, the process enters `critical_section`. On returning from `critical_section`, the process

Fig. 4.1.1.2. Differences in a Standard and Modified Invocation



passes an exit message to protector in order to notify protector that it has finished. Then the process continues executing as normal.

In a general situation, to accomplish synchronization using the approach outlined by the above scenario, it is only necessary to specify:

- i) The procedures which are to be protected by a particular guardian, and
- ii) The synchronization strategy which the protecting guardian will use.

Presumably the procedures to be protected by a particular guardian would be specified at compilation time so that the modification to the pertinent invocations could be easily made.¹ Also, presumably, a procedure could be protected by at most one guardian.

The specification of the synchronization strategy could be expressed as a function of the entire sequence of past events in a manner similar to that described in 2.6. Note that a synchronization strategy almost never uses all the information contained in the complete history of past events. Thus optimization is possible if the relevant information of the history can be encoded and if the synchronization strategy can be defined as a function of this encoding. For this reason, the definition of a strategy function will take on the following character.

First the data structure which will encode the past history is defined. The data structure must contain the following operations:

1. Note that the modifications could be made at the entry and exit code for the procedure rather than at the points of invocation.

- i) A create operation which will build an "empty" structure for use in maintaining the information about the history.
- ii) A put_request operation which will take a new request and update the data structure to reflect the fact that a new request has occurred.
- iii) A put_exit operation which will take a new exit notification and update the data structure to reflect the fact that an exit has occurred.
- iv) A put_enter operation which will update the data structure to reflect the fact that an enter has occurred.

After the data structure has been defined, the synchronization strategy must be defined on the data structure. Notice that when the strategy function returns a request that will be allowed to continue, an enter event occurs; and there must be a call to put_enter. Since put_enter and the synchronization strategy are so closely linked, they will always be combined into a single operation on the data structure. Thus the strategy function not only determines which outstanding request will be allowed to continue next, it also updates the data structure to reflect the fact that the request has been allowed to continue -- i.e., that an enter has occurred. Therefore put_enter as a separate operation is not needed since the strategy function will be responsible for updating the data structure when an enter event occurs. The strategy function is thought of as an operation on the data structure.

To summarize, recall figure 4.1.1.1. A synchronization strategy can be specified by defining a data structure which will be used to encode the past history, α . The operations put_request and put_exit take the place of the concatenation of in onto the history α , while the operation strategy takes the place of both the synchronization strategy defined on α and the concatenating of the enter events onto the history α .

This subsection has provided an overview of how synchronization might be accomplished using the simple polling guardian as a paradigm. The next subsection presents a more concrete description of this approach.

4.1.2 Definition of the Synchronization Mechanism

The synchronization mechanism will be called a *protector*. To reiterate, the purpose of a protector is to coordinate procedure calls of a program running in a multi-processing environment. The protector is defined here in terms of a particular programming language for definiteness. The language used is CLU [LISK79] because of its ability to handle data abstractions conveniently.

A new declaration statement, the *protector-create* declaration, is added to CLU. This declaration describes which procedures are to be protected -- i.e., have their invocations synchronized -- and indicates the cluster which implements the data structure (i.e., data type) that defines the synchronization strategy. The declaration creates a protector (i.e., a guardian-like procedure) and modifies all the invocations to the protected procedures.

The syntax of the protector-create declaration is given below:

create protector for procedures: idn,... using idn'

The identifiers *idn,...* are the names of the procedures whose invocations will be controlled by the protector. The procedures might be either operations defined in a cluster module or actual procedures defined in a procedure module. The protector-create declaration would appear in the module where these procedures are defined. The identifier *idn'* is a mutable data type (i.e., cluster name) which will be used to encode the past history of events and to implement the synchronization strategy.

The set of basic types of CLU is also augmented to include a new type. An object of the new type can be thought of as being a message with a very specific purpose. This purpose is to provide a means for a process executing a protected procedure to communicate with the protector which is protecting the procedure. For reasons which will become clear, this type will be named *event*. There are fundamentally two kinds of events: request and exit. A request event is a signal that is sent by a process which would like to execute a protected procedure. An exit event is a message sent to the protector when the process returns from a protected procedure. Events are created and manipulated only in the context of synchronization brought about by protectors. It is possibly confusing that within the data type event there is no subclass of messages called enter events. The reason for this absence is that a protector need not send any special message back to the process waiting to be allowed to continue execution. The protector need only activate the process. Thus no explicit mention of an enter event as a data object is ever necessary. An event can be thought of as a message containing three components: The processor id associated with the event, the type of event (either request or exit), and the procedure invocation associated with the event. Since events arise only in the context of synchronization, every event is also implicitly associated with a particular protector.¹ The operations on events permit examination of these components. Below a brief description of each operation is given:

- type_of: **proctype(event) returns(string)**
 Returns "request" or "exit" if the argument is a request event or
 an exit event respectively.
- procid: **proctype(event) returns(int)**
 Returns the unique integer which is the id of the process which
 caused the event.

1. Recall that it has been assumed that a procedure can be protected by only a single protector.

- op:** **proctype(event) returns(int)**
If the procedure associated with this event is the i^{th} procedure listed in the **protector-create** declaration, then the operation returns i .
- get_arg:** **proctype(event,int) returns(any)**
Let i be the second argument of **get_arg**. Then **get_arg** returns the i^{th} argument of the procedure call associated with this event.

The data type which encodes the past history of events (idn' above) is often called a synchronization type. A synchronization type, S , is a mutable data type which has the following operations:

- create:** **proc() returns(S)**
This operation creates an object of type S and initializes it. This object can then be used to encode relevant information about the past history of events.
- put_request:** **proc(e:event,alpha:S)**
This operation takes a new request event, e , and updates the object, α , to reflect the arrival of the event.
- put_exit:** **proc(e:event,alpha:S)**
This operation takes a new exit event, e , and updates the object, α , to reflect the arrival of the event.
- strategy:** **proc(alpha:S) returns(event) signals(null_event)**
This operation returns either a request which is to be allowed to continue and then updates S to reflect this action, or it signals **null_event** if no request is to be allowed to continue.

Briefly, the protector which is created by the declaration is a process dedicated to polling an input queue for new requests and exit events which it keeps track of by encoding the history in an object of the synchronization type. The data object is often referred to as the state of the protector. The protector examines the history encoded in

the data object and allows requesting processes -- i.e., those which are waiting -- to continue when appropriate. When a process invokes a procedure which is protected, a request event is created which is added to the end of the protector's input queue. The process then deactivates itself. The protector takes this request off its input queue and updates its state via the `put_request` operation. When the protector decides that it is safe for this process to execute the protected procedure -- i.e., when the operation strategy returns the process's request --, it performs the `allow` command which reactivates the waiting process. The operation strategy updates the protector's state to reflect that the request is now busy. Then as the process returns from the invoked procedure, it notifies the protector that this event has occurred by adding an exit event to the protector's input stream. Again the protector updates its state, this time via the `put_exit` operation. The process then continues to execute without further interference. Figure 4.1.2.3 summarizes these actions. The semantics of a protector can be given in more detail in terms of the simple polling guardian given in Chapter 2. In figure 4.1.2.4 is a procedure with the form of a polling guardian which defines the actions of a protector. This procedure is actually just a rewriting of the simple guardian procedure given in figure 2.5.9 of Chapter 2. The statement `allow(r)` is assumed to activate the procedure which is waiting for the request `r`. The data type "synchronization" is the synchronization type with which this particular protector was created. The protector's input queue is represented by `in_queue` which is treated here like a stream of events. The operation "get" removes the first event from the stream. Note that the function "strategy" can signal `null_event` in which case no activation occurs. A procedure like the one in figure 4.1.2.3 is created automatically by each protector-create declaration and is never actually seen by the programmer. He must merely supply the cluster which implements the synchronization type.

To complete the discussion of the protector, it must be shown how events are associated with the protector's execution. Recall, first, how events were associated with the operation of a polling guardian (see 2.3). The natural approach would be to say that a request event would occur with the removal of a request event from the protector's `in_queue`. Similarly, an exit event would occur when an exit event is

Fig. 4.1.2.3. Summary of Protector's Operation

Code in Main Program

```

      .
      .
create protector for procedures: op
      .          using mx;
      .
      .
op(x,y);
      .
      .

```

i) The declaration causes creation of a protector and modifies all invocations of op to cause proper creation of signals.

ii) On executing $op(x,y)$, a request event for this invocation is created and put on the end of the protector's input queue. Then the executing process deactivates itself.

iii) See iii' below.

iv) After reactivation, the process invokes $op(x,y)$. On return from op , the process creates an exit event for this invocation and puts it on the protector's input queue.

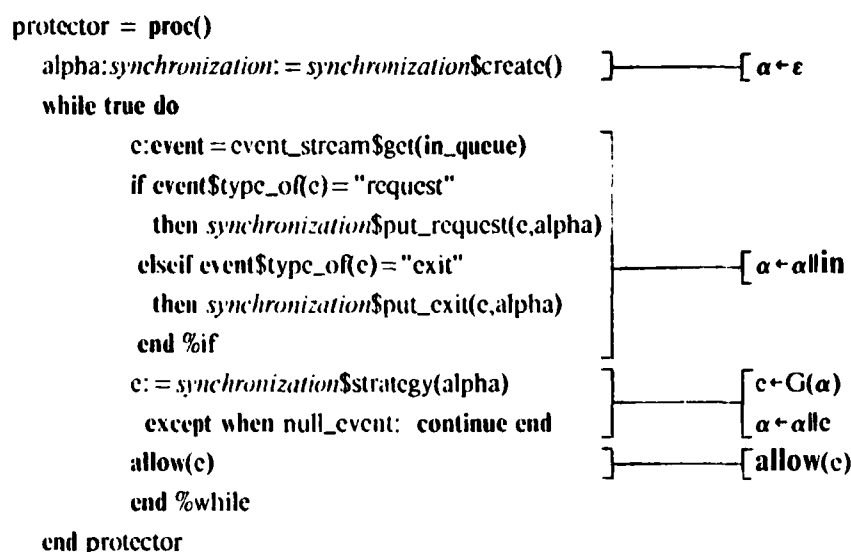
The Protector

iii') The protector gets the request; and when it eventually decides that the invocation can proceed, it activates the waiting process via the allow statement.

```

protector := proc();
  .
  .
c := event_stream$get(in_queue);
  .
  .
allow(c);
  .
  .

```


Fig. 4.1.2.4. *Semantics of a Protector*

removed from the protector's **in_queue**. Finally, an enter event would occur with the execution of the **allow** statement. However, the actual execution of the protector's procedure is hidden from the implementor of a synchronization strategy. Therefore, rather than associating the events with protector execution as above, we will instead take the approach of associating the events with the execution of the operations of the synchronization type. A request event occurs with the completion of the **put_request** operation; an exit event occurs with the completion of the **put_notice** operation; and an enter event occurs with the return from the strategy operation. After examining the protector procedure given in figure 4.1.2.4, it is clear that associating the events, as in the first approach, is equivalent to the association of events with the return from operations of the synchronization type as in the second approach.

A brief example of protector use should clarify the above description somewhat. Suppose that a programmer, in coding the procedure *cs*, realizes that it can be executed by only one process at a time. Thus a protector is needed which will force mutually exclusive access to *cs*. The module of the system containing the definition of *cs* would have the following declaration:

create protector for procedures: *cs* using *mx*.

This declaration creates a protector and modifies all invocations of *cs*. In addition to putting the above declaration in the module defining *cs*, the programmer would also code an implementation of *mx*. The cluster implementing the synchronization data type *mx* is given in figure 4.1.2.5. The implementation given in figure 4.1.2.5 uses the data type *event_seq* to manage the outstanding requests. A brief description of the

Fig. 4.1.2.5. Synchronization Type for Mutual Exclusion

```

mx = cluster is create, put_request, put_exit, strategy
  rep = record[q:event_seq,busy:bool]

  create = proc() returns(cvt)
    return(rep${q:event_seq$create(),busy:false})
  end create

  put_request = proc(e:event,alpha:cvt)
    event_seq$inq(alpha.q,e)
  end put_request

  put_exit = proc(e:event,alpha:cvt)
    alpha.busy := false
  end put_exit

  strategy := proc(alpha:cvt)returns(event)signals(null_event)
    if ~alpha.busy
      then e:event := event_seq$frst(alpha.q)
      except when empty: signal null_event end
      alpha.busy := true
      event_seq$dq(alpha.q)
      return(e)
    else signal null_event
    end %if
  end strategy
end mx

```

data type `event_seq` is included below:

- create:** **proctype() returns(event_seq)**
This operation returns an empty sequence.
- nq:** **proctype(event_seq, event)**
This operation modifies `arg1` by adding `arg2` onto the end of it.
- remove:** **proctype(event, event_seq)**
This operation removes the first occurrence of `arg1` from `arg2`.
If `arg1` does not occur in `arg2` then `arg2` remains unchanged.
- dq:** **proctype(event_seq)**
This operation removes the first event from `arg1`. If `arg1` is the empty sequence then `arg1` remains unchanged.
- frst:** **proctype(event_seq) returns(event) signals(empty)**
This operation returns the first event in `arg1`. If `arg1` is the empty sequence then it signals empty.
- frstp:** **proctype(event_seq, proctype(event) returns(bool))**
 returns(event) signals(empty)
This operation returns the first occurrence of an event in `arg1` that satisfies the predicate defined by `arg2`. If no such event exists then it signals empty.
- empty:** **proctype(event_seq) returns(bool)**
The operation returns true if `arg1` is the empty sequence and otherwise returns false.

Although many other data types besides `event_seq` can be used¹ for managing the outstanding requests, `event_seq` is often very convenient; its use will simplify the presentation of solutions.

1. For some examples of the use of other data types for managing the outstanding requests see the solutions, (figures 4.3.11 and 4.3.12), to the disk scheduler problem discussed in 4.3.

Now we return to the discussion of the cluster `mx`. The operations of `mx` should be described briefly. The `put_request` operation adds a request to the end of the event sequence `q`, which contains the rest of the requests that are currently waiting to be allowed to continue. The `put_exit` operation is called by the protector whenever a process exits from `cs`; consequently the boolean `busy` is set to false. The operation strategy checks to see if there is a process currently executing `cs` by testing the boolean `busy`. If `cs` is not busy, the first request is removed from the sequence `q` and returned so that the protector can allow it to continue. Thus the data type `mx`, in conjunction with the semantics of a protector given by the procedure defined in figure 4.1.2.4, implements the following solution:

$$F(\alpha) = \text{if } B(\alpha) = \emptyset \text{ then } \min(W(\alpha)) \\ \text{otherwise } \lambda.$$

This example might seem ridiculously complicated for something so simple as mutual exclusion; however, once the cluster `mx` has been written, the same kind of mutual exclusion can be invoked for another operation `cs1` simply with the declaration:

create protector for procedures: `cs1` using `mx`.

It is envisioned that in a user library many synchronization data types would exist, each of which would encapsulate an abstract synchronization behavior. A programmer would, as necessary, simply use these data types in protector-create declarations in order to accomplish synchronization by creating the proper protectors. If the precise kind of behavior desired were not available in the library, he would write his own synchronization type and add it to the library.

It is often helpful to think of the history which is encoded in a synchronization data type as consisting of two distinct components: information on the waiting requests; and information on the status of the resource. The `put_request` will update the waiting request component while the `put_exit` updates the resource status. If the operation strategy does not signal the `null_event`, it updates both components. In the

previous example, alpha.q was the first component while alpha.busy was the second component.

4.1.3 Implementation Issues

Above, protector creation is described as something which occurs before execution. It is assumed that when the system first starts, all the protector processes will begin executing. Making protector creation a declarative rather than an executable statement is much simpler; however, with proper care, protector creation could be made dynamic.

Recall that it has been assumed that no procedure appears in more than one protector-create declaration. It has also been tacitly assumed that none of the operations of a synchronization type ever appear in a protector-create declaration. That these assumptions are met, is easily checked automatically before execution time if protector creation is not dynamic.

Although the implementation of the protector's input queue has not been specified, it is possible that some sort of low level synchronization might be necessary among the processes adding events to the end of the queue and the protector's process which is removing requests from the queue. Rather than elaborate on how this might be accomplished, it will simply be assumed that the adding and removing of an event from the protector's input queue is an atomic operation; thus no problems arise with maintaining queue consistency.

The speed of a protector's process relative to the arrival rate of events and relative to the speed of the requesting processes is important, not to resource consistency, but to the viability of the handling of priority constraints and to the general practicality of the protector. The input queue of a protector should never contain many events in order to force the guardian's state to represent as closely as possible those requests which are actually waiting. The protector's process must run at a high speed relative to the arrival rate of events in order to ensure that a protector's

input queue really does remain relatively empty. Since an event is taken off the queue for each iteration of the protector, the fastest burst rate at which events can occur must be less than the longest time it takes the process to update the protector's state twice and to evaluate the strategy function once. The speed of the protector's process should be at least as fast as the the fastest of the requesters. Otherwise performance of high speed processes would be degraded.

Since it has been argued that the protector's process must run at a relatively high priority, it is important not to waste processing resources needlessly with pointless polling. Fortunately, it is easy to prevent the protector's process from looping uselessly, looking for new events. This is done by deactivating the protector's process if the input queue is ever empty after a call to strategy has just signaled `null_event`. The protector's process remains deactivated until a new event is put on the input queue, at which time it is reactivated. In this way the polling is "conceptual" only and is not wasteful of resources.

4.2 Readers/Writers Problem

The readers/writers problem has become one of the most commonly discussed examples of a synchronization problem. Since it was first introduced in [COUR71], many solutions have been proposed and many synchronization mechanisms have been justified by demonstrating elegant implementations of these solutions. The problem can be described as follows: Suppose there is a data base which various users must access. An access may be either the reading of a part of the data base or the writing of some new information into the data base. In either case it is assumed that the accesses are not necessarily atomic. Thus in order to preserve the consistency of the operations and the data base, reads must not be allowed concurrently with writes, and concurrent writes must not be allowed. Basically any solution to the predicate

$$(0) \quad Mx(r[\text{write}], r')$$

is a solution to the general readers/writers problem. There are, however, many

versions of the problem, some of which are discussed below.

One version of the problem [BLOO79] requires that each request be served in a first come first served order. This can be expressed more formally by the predicate

$$(1) \quad Mx(r[\text{write}], r') \wedge \text{FIFO}(r).$$

The cluster described in figure 4.2.6 defines a protector which implements an optimal solution to this version of the problem. It is assumed that in the module defining the procedures read and write, there is the declaration

Fig. 4.2.6. FCFS Solution to Readers/Writers Problem

```

rw = cluster is create, put_request, put_exit, strategy
  read = 1
  write = 2
  rep = record[q: event_seq, writes, reads: int]

  create = proc() returns(cvt)
    return(rep$ {q: event_seq$create(), writes: 0, reads: 0})
  end create

  put_request = proc(c: event, a: cvt)
    event_seq$enq(a.q, c)
  end put_request

  put_exit = proc(c: event, a: cvt)
    if event$op(c) = write then a.writes := a.writes + 1
    elseif event$op(c) = read then a.reads := a.reads + 1 end
  end put_exit

  strategy = proc(a: cvt) returns(event) signals(null_event)
    c: event := event_seq$frst(a.q)
    except when empty: signal null_event end
    if a.writes = 0 & (a.reads = 0) | event$op(c) = read
      then event_seq$dq(a.q)
      if event$op(c) = write then a.writes := a.writes + 1
      elseif event$op(c) = read then a.reads := a.reads + 1 end
      return(c)
    end %if
    signal null_event
  end strategy
end rw

```

create protector for procedures: read, write using rw.

Briefly, $a.q$ is a sequence of events which contains the requests which are currently outstanding. The integer variables $a.reads$ and $a.writes$ are the number of currently busy reads and writes respectively. The strategy here is to take the oldest outstanding request and see if it can be allowed to continue. If it can, then the request is returned; otherwise, $null_event$ is signaled.

Although this solution is an optimal solution to (1), it is not an optimal solution to (0). In figure 4.2.7 is a cluster which implements an optimal solution to (0).¹ This solution is the same as the one given by Andrews [ANDR79]. The strategy here is first to determine what kind of requests can be allowed to continue and then to choose the first outstanding request of that kind. Another optimal solution to (0) is defined by the cluster in figure 4.2.8. This solution satisfies the readers priority version of the readers/writers problem mentioned in both [BLOO79] and [GREI76]. This problem is characterized by the following predicate:

$$(2) \quad Mx(r[write], r') \wedge Pr(read, write).$$

Another very similar problem, also mentioned in [BLOO79] and [GREI76], is the writers priority version. This version can be formalized by the predicate:

$$(3) \quad Mx(r[write], r') \wedge Pr(write, read).$$

Figure 4.2.9 presents an implementation of a solution to (3). The similarity between this implementation and the previous is obvious.

1. For a proof of this claim see subsection 4.6.4.

Fig. 4.2.7. Weak Readers Priority Solution to the Readers/Writers Problem

```

rw = cluster is create, put_request, put_exit, strategy
  read = 1
  write = 2
  rep = record[q:event_seq,writes,reads:int]

  create = proc() returns(cvt)
    return(rep${q:event_seq$create(),writes:0,reads:0})
  end create

  put_request = proc(c:event,a:cvt)
    event_seq$enq(a,q,c)
  end put_request

  put_exit = proc(c:event,a:cvt)
    if event$op(c) = write then a.writes: = a.writes-1
    elseif event$op(c) = read then a.reads: = a.reads-1 end
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    if (a.reads=0)&(a.writes=0)
      then c:event: = event_seq$frst(a,q)
        if event$op(c) = write then a.writes: = a.writes + 1
        elseif event$op(c) = read then a.reads: = a.reads + 1 end
        event_seq$dq(a,q)
        return(c)
      elseif a.writes=0
        then c:event: = event_seq$frstp(a,q,readers)
          a.reads: = a.reads + 1
          event_seq$remove(c,a,q)
          return(c)
        end %if
      except when empty: signal null_event end
    signal null_event
  end strategy

  readers = proc(c:event) returns(bool)
    return(event$op(c) = read)
  end readers

end rw

```

Fig. 4.2.8. Readers Priority Solution to the Readers/Writers Problem

```

rw = cluster is create, put_request, put_exit, strategy
  read = 1
  write = 2
  rep = record[q:event_seq,writes,reads:int]

  create = proc() returns(cvt)
    return(rep[q:event_seq$create(),writes:0,reads:0])
  end create

  put_request = proc(c:event,a:cvt)
    event_seq$enq(a.q,e)
  end put_request

  put_exit = proc(c:event,a:cvt)
    if event$op(c) = write then a.writes := a.writes-1
    elseif event$op(c) = read then a.reads := a.reads-1 end
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    c:event := event_seq$firstp(a.q,readers)
    except when empty:
      c := event_seq$firstp(a.q,writers)
      except when empty: signal null_event end
      if (a.reads = 0) & (a.writes = 0)
        then event_seq$remove(c,a.q)
        a.writes := a.writes + 1
        return(c)
      end %if
    end %except
    if a.writes = 0
      then event_seq$remove(c,a.q)
      a.reads := a.reads + 1
      return(c)
    end %if
    signal null_event
  end strategy

  readers = proc(c:event) returns(bool)
    return(event$op(c) = read)
  end readers

  writers = proc(c:event) returns(bool)
    return(event$op(c) = write)
  end writers

end rw

```

Fig. 4.2.9. Writers Priority Solution to the Readers/Writers Problem

```

rw = cluster is create, put_request, put_exit, strategy
  read = 1
  write = 2
  rep = record[q:event_seq,writes,reads:int]

  create = proc() returns(evt)
    return(rep[q:event_seq$create(),writes:0,reads:0])
  end create

  put_request = proc(e:event,a:evt)
    event_seq$enq(a,q,e)
  end put_request

  put_exit = proc(e:event,a:evt)
    if event$op(e) = write then a.writes = a.writes-1
    elseif event$op(e) = read then a.reads = a.reads-1 end
  end put_exit

  strategy = proc(a:evt) returns(event) signals(null_event)
    e:event := event_seq$firstp(a.q,writers)
    except when empty:
      e := event_seq$firstp(a.q,readers)
    except when empty: signal null_event end
    if a.writes = 0
      then event_seq$remove(e,a,q)
      a.reads = a.reads + 1
      return(e)
    end %if
  end %except
  if (a.reads = 0) & (a.writes = 0)
    then event_seq$remove(e,a,q)
    a.writes = a.writes + 1
    return(e)
  end %if
  signal null_event
end strategy

readers = proc(e:event) returns(bool)
  return(event$op(e) = read)
end readers

writers = proc(e:event) returns(bool)
  return(event$op(e) = write)
end writers

end rw

```

Notice that none of the solutions presented thus far, except for the first, is fair to both readers and writers. Below, a protector which satisfies the following version of the readers/writers problem is described:

$$(4) \quad Mx(r[\text{write}], r') \wedge Fr(r).$$

The cluster in figure 4.2.10 implements a solution which is described in both [HOAR74] and [BLOO79]. In this solution, if there are writers waiting when a read is requested, the read must wait until one write completes. When a write terminates, then all waiting reads may proceed. In the implementation, $a.frq$ contains the outstanding reads which either were requested when no writes were active or had seen a write complete while they were waiting. The event sequence $a.rq$ is the outstanding reads which are waiting for a write to complete. The sequence $a.wq$ is the outstanding writes. For a proof that the protector does indeed satisfy (4), see 4.6.2.

Fig. 4.2.10. Fair Solution to the Readers Priority Readers/Writers Problem

```

rw = cluster is create, put_request, put_exit, strategy
  read = 1
  write = 2
  rep = record[rq,wq,frq:event_seq,writes,reads:int]

  create = proc() returns(cvt)
    return(rep${rq:event_seq$create(),wq:event_seq$create(),frq:event_seq$create(),
      writes:0,reads:0})
  end create

  put_request = proc(c:event,a:cvt)
    if event$op(c) = read
      then if event_seq$empty(a.wq)&a.writes = 0 then event_seq$enq(a.frq,c)
        else event_seq$enq(a.rq,c)
      end %if
    elseif event$op(c) = write
      then event_seq$enq(a.wq,c)
    end %if
  end put_request

  put_exit = proc(c:event,a:cvt)
    if event$op(c) = read
      then a.reads := a.reads + 1
    elseif event$op(c) = write
      then a.writes := a.writes + 1
        a.frq := a.rq
        a.rq := event_seq$create()
    end %if
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    if ~event_seq$empty(a.frq)&a.writes = 0
      then c:event := event_seq$frst(a.frq)
        event_seq$dq(a.frq)
        a.reads := a.reads + 1
        return(c)
    elseif ~event_seq$empty(a.wq)&((a.reads = 0)&(a.writes = 0))
      then c:event := event_seq$frst(a.wq)
        event_seq$dq(a.wq)
        a.writes := a.writes + 1
        return(c)
    end %if
    signal null_event
  end strategy
end rw

```

4.3 Disk Scheduler Problem

In this section the disk scheduling problem is discussed. The problem is to order accesses to a single movable head disk in an attempt to optimize system efficiency and individual response time [TEOR72]. When accessing a movable head disk, by far the largest part of the delay is due to head movement. Thus solutions to this problem employ an algorithm for arranging the outstanding requests so as to minimize head movement. Two such algorithms are discussed below: the SCAN or elevator algorithm; and the C-SCAN or circular scan algorithm. Each approach has its advantages and disadvantages depending on whether disk utilization is heavy or light. Note that any implementation of either algorithm must prevent concurrent accesses to the disk since the disk can handle only one request at a time.

In the SCAN algorithm the head moves or scans across the disk in one direction, servicing requests as it goes until there are no more outstanding requests for cylinders beyond the head's current position in the direction it is moving. Then, assuming there are outstanding requests back in the other direction, the head reverses direction and moves back across the disk servicing requests as it goes. Figure 4.3.11 presents a cluster which implements this approach. This solution is similar to one described in [HOAR74] except that this solution satisfies $Fr(r)$ whereas Hoare's solution does not. It is assumed here that somewhere a procedure "disk_access" is defined and that in its defining module is the declaration:

create protector for procedures: disk_access using ds.

It is also assumed that the first argument of "disk_access" specifies the cylinder address of the access. In the implementation, the set of outstanding requests is partitioned between two priority queues:¹ a.uq and a.lq. The requests are sorted on the priority queues according to what cylinder the request is accessing. The priority queue a.lq

1. For a nice implementation of priority queues using heaps see [LISK79], pp137-139.

Fig. 4.3.11. The SCAN Solution to the Disk Scheduler Problem

```

ds = cluster is create, put_request, put_exit, strategy
  up_scan = 1
  down_scan = 2
  rep = record[lq,uq:p_queue[event],pos,dir:int,busy:bool]

  create = proc() returns(cvt)
    return(rep$ {lq:p_queue[event]$create(descending),
                uq:p_queue[event]$create(ascending),
                pos:0,dir:0,busy:false})
  end create

  put_request = proc(c:event,a:cvt)
    arg:int := event$get_arg(c,1)
    if arg > a.pos then p_queue[event]$insert(a.uq,c)
    elseif arg < a.pos then p_queue[event]$insert(a.lq,c)
    elseif arg = a.pos then
      if a.dir = up_scan then p_queue[event]$insert(a.lq,c)
      elseif a.dir = down_scan then p_queue[event]$insert(a.uq,c) end
    end %if
  end put_request

  put_exit = proc(c:event,a:cvt)
    a.busy := false
    a.pos := event$get_arg(c,1)
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    c:event
    if ~a.busy & (~p_queue[event]$empty(a.lq) | ~p_queue[event]$empty(a.uq))
      then if (a.dir = up_scan) & (p_queue[event]$empty(a.lq))
        then a.dir := down_scan
        elseif (a.dir = down_scan) & (p_queue[event]$empty(a.uq))
        then a.dir := up_scan end
      if a.dir = up_scan then c := p_queue[event]$remove(a.uq)
      elseif a.dir = down_scan then c := p_queue[event]$remove(a.lq) end
      a.busy := true
      return(c)
    end %if
    signal null_event
  end strategy

  descending = proc(c1,c2:event) returns(bool)
    return(event$get_arg(c2,1) < event$get_arg(c1,1))
  end descending

  ascending = proc(c1,c2:event) returns(bool)
    return(event$get_arg(c1,1) < event$get_arg(c2,1))
  end ascending

end ds

```

contains its requests in descending order while the priority queue `a.uq` contains its requests in ascending order. The integer variable `a.pos` is the cylinder address where the head is currently located. The integer variable `a.dir` is the direction in which the head is currently scanning. The direction determines from which queue requests will be taken in the operation strategy. When `dir=up_scan`, then requests are removed from `a.uq`; and when `dir=down_scan`, then requests are removed from `a.lq`. The operation strategy changes the direction whenever the queue in the current direction is empty and the queue in the other direction is not empty. When `put_request` is called, the request passed is inserted into `a.uq`, the upper queue, if the cylinder address of the access is greater than the current position. If on the other hand the address is less than the current position, the request is inserted into `a.lq`, the lower queue. When the access address of the request is the current position, then the request is put on the queue from whichever strategy currently is not removing requests. This prevents starvation of other requests.

In the C-SCAN algorithm the head moves across the disk servicing requests in only one direction. When there are no more outstanding requests in this direction, then, assuming there are still some outstanding requests, the head is moved all the way back across the disk to service the request with the cylinder address which is farthest from the current position. Figure 4.3.12 gives an implementation of this approach. This solution is similar to one proposed in [ANDR79]. The outstanding requests are partitioned between two priority queues, `a.uq` and `a.tq`. Requests in each of these queues are sorted in ascending order by cylinder address. The integer `a.pos` is the cylinder where the head is currently located. When `put_request` is called, the request is inserted into the upper queue (i.e., `a.uq`) if it accesses a cylinder greater than the current position. Otherwise the request is put on the temporary queue, `a.tq`. In strategy, requests are removed from `a.uq`. When `a.uq` is empty, then if there are still outstanding requests, `a.uq` is set equal to `a.tq` and `a.tq` is re-initialized as an empty queue.

Fig. 4.3.12. The CSCAN Solution to the Disk Scheduler Problem

```

ds = cluster is create, put_request, put_exit, strategy
  rep = record[uq,tq:p_queue[event],pos:int,busy:bool]

  create = proc() returns(cvt)
    return(rep$ {uq:p_queue[event]$create(ascending),
               tq:p_queue[event]$create(ascending),
               pos:0,busy:false})
  end create

  put_request = proc(e:event,a:cvt)
    if event$get_arg(e,1) > a.pos then p_queue[event]$insert(a.uq,e)
    elseif event$get_arg(e,1) <= a.pos then p_queue[event]$insert(a.tq,e) end
  end put_request

  put_exit = proc(e:event,a:cvt)
    a.busy = false
    a.pos = event$get_arg(e,1)
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    if ~a.busy & (~p_queue[event]$empty(a.uq) | ~p_queue[event]$empty(a.tq))
      then if ~p_queue[event]$empty(a.uq)
        then a.uq = a.tq
        a.tq = p_queue[event]$create(ascending)
        end %if
      c:event = p_queue[event]$remove(a.uq)
      a.busy = true
      return(c)
    end %if
    signal null_event
  end strategy

  ascending = proc(c1,c2:event) returns(bool)
    return(event$get_arg(c1,1) < event$get_arg(c2,1))
  end ascending
end ds

```

4.4 Five Dining Philosophers

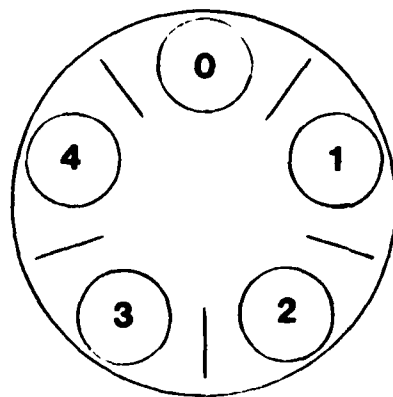
This section examines the problem of the five dining philosophers [DIJK71]. In this problem there are n Chinese philosophers or sages which alternate between thinking and eating.¹ The dining table at which they eat is set so that each sage has his own place at a round table. Unfortunately, there are only n chop sticks which are arranged so that between each two place settings there is a single chop stick. See figure 4.4.13. Thus philosophers with adjacent place settings cannot eat simultaneously. If the sages are numbered 0 through $(n-1)$ then this constraint can be restated as

$$(0) \quad (i-j \equiv \pm 1) \bmod n \rightarrow Mx(r[i], r[j]).$$

In transforming this problem into a practical programming problem, it is assumed that a procedure which is named "eat" is called by various processes numbered 0 through $(n-1)$. In the header of the module where eat is defined, the following declaration appears:

create protector for procedures: eat using dp[5].

Fig. 4.4.13. The Dining Philosophers Problem



1. In Dijkstra's original statement of the problem, $n = 5$.

Figure 4.4.14 presents a cluster which defines an implementation which is an optimal solution to (0). In the cluster, `a.q` contains the outstanding requests and the array `a.table` tells who is currently eating. Thus `a.table[i] = 1` if and only if philosopher `i` is eating and otherwise `a.table[i] = 0`. The procedure "bind" needs some explanation. An invocation of `bind(p,i,s)`, where `p` is a function with `k` arguments, returns the function `p'` with `k-1` arguments which results from the binding of `s` to the i^{th} argument of `p`. Thus `bind` permits partial parameterization as in some extensions of ALGOL 68, [LIND74] and [LIND76].

Fig. 4.4.14. Optimal Solution to the Dining Philosophers Problem

```

dp = cluster[n:int] is create, put_request, put_exit, strategy
  rep = record[q:event_seq,table:array[int]]

  create = proc() returns(cvt)
    return(rep$[q:event_seq$create(),table:array[int]$fill(0,n-1,0)])
  end create

  put_request = proc(e:event,a:cvt)
    event_seq$inq(a.q,e)
  end put_request

  put_exit = proc(e:event,a:cvt)
    a.table[event$procid(e)] := 0
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    e:event := event_seq$firstp(a.q,bind(open,1,a.table))
    except when empty: signal null_event end
    a.table[event$procid(e)] := 1
    event_seq$remove(e,a.q)
    return(e)
  end strategy

  open = proc(tbl:array[int],e:event) returns(bool)
    i:int := event$procid(e)
    return((tbl[(i+1)/n] = 0) & (tbl[(i-1)/n] = 0))
  end open

end dp

```

As is pointed out in [DIJK71], the above solution may result in the starvation of one or more of the sages. A possible version of the dining philosophers problem also requires that the solution be fair to all requests. More formally:

$$(1) \quad [(i-j) \equiv \pm 1 \bmod n \rightarrow Mx(r[i], r[j])] \wedge Fr(r).$$

Figure 4.4.15 gives a cluster which results in the requests being served in a first come first served order. This will result in very little parallelism. For instance, if the oldest outstanding request -- i.e., the first one on the queue -- is blocked from eating, then even though all the rest of the outstanding requests might be free to eat, they cannot.

Fig. 4.4.15. FCFS Solution to the Dining Philosophers Problem

```

dp = cluster[n:int] is create, put_request, put_exit, strategy
  rep = record[q:event_seq, table:array[int]]

  create = proc() returns(cvt)
    return(rep[q:event_seq$create(), table:array[int]$fill(0, n-1, 0)])
  end create

  put_request = proc(e:event, a:cvt)
    event_seq$enq(a.q, e)
  end put_request

  put_exit = proc(e:event, a:cvt)
    a.table[event$procid(e)] := 0
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    c:event := event_seq$frst(a.q)
    except when empty: signal null_event end
    j:int := event$procid(c)
    if (a.table[(j+1)//n] = 0) & (a.table[(j-1)//n] = 0)
      then a.table[j] := 1
      event_seq$dq(a.q)
      return(c)
    end %if
    signal null_event
  end strategy
end dp

```

A better approach to a solution to (1) is suggested by Dijkstra in [DIJK71]. This approach introduces the notion of very hungry. One way to formalize this notion is as follows: A sage is hungry when his request to eat is outstanding, and a sage is very hungry if while he has been hungry, k other sages have finished eating. Very hungry sages are served in a FCFS manner and given absolute priority over sages that are simply hungry.¹ Figure 4.4.16 gives an implementation of such a solution. The integer array, `a.status`, maintains the status of the sages as follows:

<code>a.status[i] = 0</code>	Sage i is not active.
<code>a.status[i] = -1</code>	Sage i is eating
<code>a.status[i] = m > 0</code>	Sage i is hungry and $m-1$ sages have eaten since i has been hungry. If $m > k$, then i is very hungry.

All outstanding requests are kept in the event sequence `a.q`. The predicate `vh` is true of an outstanding request when it is both very hungry and neither of its adjacent neighbors is eating. The predicate `h` is true of an outstanding request when neither of its adjacent neighbors is eating. Note that we have made k a parameter which would be passed by the protector-create declaration.

¹ This is the FCFS solution given above.

Fig. 4.4.16. Fair Solution to Dining Philosophers Problem with Very Hungry Sages

```

dp = cluster[n:int,k:int] is create, put_request, put_exit, strategy
  rep = record[q:event_seq,status:array[int]]

  create = proc() returns(cvt)
    return(rep${q:event_seq$create(),status:array[int]$fill(0,n-1,0)})
  end create

  put_request = proc(e:event,a:cvt)
    a.status[event$procid(e)] := 1
    event_seq$enq(a.q,e)
  end put_request

  put_exit = proc(e:event,a:cvt)
    j:int := 0
    while j < n do
      if a.status[j] > 0 then a.status[j] := a.status[j] + 1 end
      j := j + 1
    end %while
    a.status[event$procid(e)] := 0
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    c:event := event_seq$frstp(a.q,bind(vh,1,a.status))
    except when empty:
      c := event_seq$frstp(a.q,bind(h,1,a.status))
      except when empty: signal null_event end
    end %except
    a.status[event$procid(e)] := -1
    event_seq$remove(c,a.q)
    return(c)
  end strategy

  vh = proc(st:array[int],e:event) returns(bool)
    return(h(st,e)&st[event$procid(e)] >= k)
  end vh

  h = proc(st:array[int],e:event) returns(bool)
    i:int := event$procid(e)
    return((st[(i+1)//n] ~ -1)&(st[(i-1)//n] ~ -1))
  end h

end dp

```

4.5 Bounded Buffer Problem

In the bounded buffer problem it is assumed that there is a buffer which can contain, at most, "max" items of information. A group of processes called producers deposits information in the buffer, an item at a time, by invoking a procedure named "produce." Another group of processes called consumers removes information from the buffer an item at a time by invoking a procedure called "consume." The problem is to synchronize calls to produce and calls to consume so that there is not mutual interference. Also when an invocation of produce is allowed, the buffer must not be full; and when an invocation of consume is allowed, the buffer must not be empty. The predicate $Z(\alpha)$ below captures the fact that in order for consume to proceed, the buffer must not be empty.

$$Z(\alpha) \equiv \forall \beta [\beta \prec \alpha \rightarrow (0 \leq \#_{\beta} x[\text{produce}] - \#_{\beta} c[\text{consume}])]^1$$

where $\#_{\beta} t[o]$ is the number of events in β with operation o and of type t .

The predicate $M(\alpha)$ captures formally that in order for produce to proceed, the buffer must not be full.

$$M(\alpha) \equiv \forall \beta [\beta \prec \alpha \rightarrow (\#_{\beta} c[\text{produce}] - \#_{\beta} x[\text{consume}] \leq \text{max})]$$

If no assumptions are made about the structure of the buffer, a solution must prevent simultaneous invocations of produce and consume. Naturally two separate invocations of either produce or consume cannot be allowed to occur simultaneously. Thus the following predicate defines the most general version of the bounded buffer problem:

$$(0) \quad Z \wedge M \wedge Mx(r, r').$$

In figure 4.5.17 is a cluster which defines a protector that implements a solution to (0). It is assumed that in the module where the procedures produce and consume are

1. Recall from 2.2.1 that $\beta \prec \alpha$ means that β is a finite initial segment of α .

Fig. 4.5.17. Nearly FIFO Solution to the Bounded Buffer Problem

```

hh = cluster[1: max: int] is create, put_request, put_exit, strategy
  produce = 1
  consume = 2
  rep = record[q: event_seq, count: int, busy: bool]

  create = proc() returns(cvt)
    return(rep$(q: event_seq$(create()), count: 0, busy: false))
  end create

  put_request = proc(c: event, a: cvt)
    event_seq$(nq(a.q, c))
  end put_request

  put_exit = proc(c: event, a: cvt)
    a.busy := false
    if event$(op(c)) = produce then a.count := a.count + 1
    elseif event$(op(c)) = consume then a.count := a.count - 1 end %if
  end put_exit

  strategy = proc(a: cvt) returns(event) signals(null_event)
    if ~a.busy
      then c: event := event_seq$(frst(a.q))
        if event$(op(c)) = produce & (a.count >= max)
          then c := event_seq$(frstp(a.q, consumer))
        elseif event$(op(c)) = consume & (a.count <= max)
          then c := event_seq$(frstp(a.q, producer)) end %if
        event_seq$(remove(c, a.q))
        a.busy := true
        return(c)
      end %if
    except when empty: signal null_event end
  signal null_event
  end strategy

  consumer = proc(c: event) returns(bool)
    return(event$(op(c)) = consume)
  end consumer

  producer = proc(c: event) returns(bool)
    return(event$(op(c)) = produce)
  end producer

end hh

```


defined, the following declaration appears:

create protector for procedures: produce,consume using bb.

The solution being implemented here attempts to serve requests on a first come first served basis. Sometimes in order to prevent deadlock this is not possible. For example, if a request to consume were the first outstanding request but the buffer were empty, then the strategy would return the first outstanding produce request, assuming the buffer were not busy. In the implementation, the boolean `a.busy` is true whenever either a production or a consumption is in progress; otherwise it is false. All outstanding requests are kept in the event sequence `a.q`. The integer `a.count` is the number of items currently in the buffer.

Another version of the bounded buffer problem assumes that produce and consume can be executed concurrently. This might be the case if, for example, the buffer were implemented as a simple array. This version of the problem is made explicit by the predicate

$$(1) \quad Z \wedge M \wedge Mx(\text{produce}, \text{produce}) \wedge Mx(\text{consume}, \text{consume}).$$

Figure 4.5.18 presents an implementation of a solution to (1). The boolean `a.c_busy` is true if and only if a process is currently executing consume. Similarly `a.p_busy` is true if and only if a process is currently executing produce. The outstanding requests to produce are kept in `a.pq` while the outstanding requests to consume are kept in `a.cq`. The integer variable `a.count` is the number of items in the buffer. For a proof of the correctness of this implementation see 4.6.3.

Fig. 4.5.18. Solution to the Bounded Buffer Problem

```

bb = cluster[max:int] is create, put_request, put_exit, strategy
  produce = 1
  consume = 2
  rep = record[pq:cq:event_seq,count:int,p_busy,c_busy:bool]

  create = proc() returns(cvt)
    return(rep{$pq:event_seq$create(),cq:event_seq$create(),
      count:0,p_busy:false,c_busy:false})
  end create

  put_request = proc(c:event,a:cvt)
    if event$op(c) = produce then event_seq$enq(a.pq,c)
    elseif event$op(c) = consume then event_seq$enq(a.cq,c) end %if
  end put_request

  put_exit = proc(c:event,a:cvt)
    if event$op(c) = produce
      then a.count := a.count + 1
      a.p_busy := false
    elseif event$op(c) = consume
      then a.count := a.count - 1
      a.c_busy := false
    end %if
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
    if ~a.c_busy & (0 < a.count) & ~event_seq$empty(a.cq)
      then c:event := event_seq$frst(a.cq)
      event_seq$ddq(a.cq)
      a.c_busy := true
      return(c)
    elseif ~a.p_busy & (a.count < max) & ~event_seq$empty(a.pq)
      then c:event := event_seq$frst(a.pq)
      event_seq$ddq(a.pq)
      a.p_busy := true
      return(c)
    end %if
    signal null_event
  end strategy
end bb

```

4.6 Proof of Implementation Correctness

This section is devoted to the development of a methodology for verifying the correctness of implementations. The way in which protectors have been defined makes it possible to use the basic techniques of the Floyd-Hoare partial correctness method [FLOY67] in order to verify many properties of a protector. Some properties require a different approach. Often these properties can be verified using the intermittent assertion method proposed in [BURS74] and [MANN78]. In the first subsection a brief outline of an approach to verifying correctness is given. Then several examples are examined.

Before giving an outline of the approach it is useful to clarify the goal of the methodology. The goal is to make possible the proof that the behavior of the guardian as implemented by the protector is a subset of the set of sequences satisfying a problem specification. In order to accomplish this goal, the methodology first shows that at any given point in execution, the protector's state, as represented by an object of its synchronization type, accurately encodes all the important information about the history of events which have occurred thus far. Then the methodology must show that the state of the protector implies both that nothing bad happens and that certain good things do happen. Thus the goals here are somewhat different from those of a methodology for verifying the correctness of a standard sequential program. Such a methodology need only prove that, given an input, the program will terminate with a given output. There is no question of behavior over continued operation.

4.6.1 An Outline of the Methodology

Following Lamport [LAMP78] the methodology separates specifications into two categories:

- i) Safety properties -- i.e., those specifications which state that something bad cannot happen; and
- ii) Liveness properties -- i.e., those specifications which state that something good must happen.

In general, consistency and priority constraints are examples of safety properties while fairness constraints are examples of liveness properties.

Definition 4.6.1.1: A *behavioral invariant* is a predicate which is always true after the execution of the synchronization create operation whenever the protector's process is not executing an operation of its synchronization type.

Because of theorem 3.3.3, we can prove that a protector satisfies a continuous predicate P by proving that P is a behavioral invariant. This is accomplished by showing that P holds immediately after the execution of the create operation and that for each additional synchronization operation, if P holds before its execution, it will hold after its execution. If P is a simple predicate, then by theorem 3.4.3 it is sufficient to prove that P is true after the create operation and that P remains invariant with the execution of strategy. Sometimes it is useful to restate a predicate P in the form $\forall \beta [\beta \prec \alpha \rightarrow Q(\beta)]$ where Q is a predicate that tests only information which is encoded by the synchronization type. By theorem 3.4.4, P can be shown to be a behavioral invariant simply by showing that Q is true after the create operation and that Q remains invariant with the execution of strategy. Notice that behavioral invariants are very similar to the invariant assertions of the Floyd-Hoare method [FLOYD67] for establishing partial correctness of simple sequential programs and can therefore be proved by the same techniques.

To establish that a protector satisfies a liveness property is more difficult since liveness properties are not continuous. Instead of establishing behavioral invariants, we must prove predicates of the form: "If the point of execution is at point L of the cluster with state Q , then eventually control will be at point L' with state Q' ." These assertions are proved by induction on the state of the protector. This technique is

precisely the method of intermittent assertions which is put forth in [BURS74] and [MANN78].

Below we give a brief outline of how to establish the correctness of a protector:

- A) First the specification is decomposed into its component predicates.
- B) The following is done for each.
 - 1) Rephrase the predicate in terms of the protector state.
 - 2) Establish that the portions of the state mentioned in the rephrasing actually encode the proper information about the past history of events.
 - 3) Depending on whether or not the predicate is continuous, do one of the following.
 - a) If it is continuous, prove it to be a behavioral invariant.
 - b) If it is not continuous, prove it true using the techniques of intermittent assertions.

In the following subsections we examine several examples in which the above outline is followed.

4.6.2 Correctness of a Solution to the Readers/Writers Problem

In this subsection, the protector described in figure 4.6.2.19 is examined and is proved to be a solution¹ to the predicate

1. This implementation was discussed previously in 4.2.

$$(0) \quad Mx(r[\text{write}],r') \wedge Fr(r).$$

We start by examining the predicate $Mx(r[\text{write}],r')$. Note that it can be rewritten as

$$\forall \beta [\beta < \alpha \rightarrow ((|B(\beta, \text{write})| = 0 \vee |B(\beta, \text{read})| = 0) \wedge |B(\beta, \text{write})| \leq 1)].^1$$

From theorem 3.4.4 we know that we need only verify that

$$(1) \quad ((|B(\alpha, \text{write})| = 0 \vee |B(\alpha, \text{read})| = 0) \wedge |B(\alpha, \text{write})| \leq 1)$$

is true after the execution of create and is an invariant of the execution of strategy.

An examination of the cluster rw reveals that a.reads and a.writes presumably are the number of busy read and write requests respectively. Thus the predicate (1) can be rewritten in terms of the protector's state as

$$(2) \quad [(a.\text{reads} = 0) \vee (a.\text{writes} = 0)] \wedge [a.\text{writes} \leq 1].$$

In order to show that (2) is actually the same as (1) it must be shown that a.reads and a.writes actually do represent the number of busy requests -- i.e., that the following are behavioral invariants:

$$(3) \quad a.\text{reads} = \# e[\text{read}] - \# x[\text{read}]$$

$$(4) \quad a.\text{writes} = \# e[\text{write}] - \# x[\text{write}]$$

Further examination of rw reveals that the event sequences a.rq and a.frq must contain the outstanding read requests and a.wq must contain the outstanding write requests. To make certain of this, the following predicates must be shown to be behavioral invariants:

1. For a finite set A, |A| is the number of elements in A.

Fig. 4.6.2.19. Fair Solution to the Readers Priority Readers/Writers Problem

```

rw = cluster is create, put_request, put_exit, strategy
  read = 1
  write = 2
  rep = record[rq,wq,frq:event_seq,writes,reads:int]

  create = proc() returns(cvt)
1    return(rep${rq:event_seq$create(),wq:event_seq$create(),frq:event_seq$create(),
               writes:0,reads:0})
    end create

  put_request = proc(c:event,a:cvt)
2    if event$op(c) = read
3      then if event_seq$empty(a.wq)&a.writes=0 then event_seq$enq(a.frq,c)
4            else event_seq$enq(a.rq,c)
              end %if
5    elseif event$op(c) = write
6      then event_seq$enq(a.wq,c)
        end %if
    end put_request

  put_exit = proc(c:event,a:cvt)
7    if event$op(c) = read
8      then a.reads: = a.reads+1
9    elseif event$op(c) = write
10     then a.writes: = a.writes+1
11         a.frq: = a.rq
12         a.rq: = event_seq$create()
    end %if
  end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
13    if ~event_seq$empty(a.frq)&a.writes=0
14      then c:event: = event_seq$first(a.frq)
15          event_seq$dq(a.frq)
16          a.reads: = a.reads+1
17          return(c)
18    elseif ~event_seq$empty(a.wq)&((a.reads=0)&(a.writes=0))
19      then c:event: = event_seq$first(a.wq)
20          event_seq$dq(a.wq)
21          a.writes: = a.writes+1
22          return(c)
    end %if
23    signal null_event
  end strategy
end rw

```

$$(5) \quad (a.rq) \cup (a.frq) = W(\alpha, \text{reads})^1$$

$$(6) \quad (a.wq) = W(\alpha, \text{writes})$$

Now if we assume that every process which invokes either read or write will eventually return, the predicate $Fr(r)$ can be rephrased as follows:

(7) "If a request r is ever on $a.rq$, $a.frq$, or $a.wq$, then eventually r will be removed and returned by the operation strategy."

With this introduction, we continue in more detail. First we prove that (3) is a behavioral invariant. Surely (3) is true immediately after the execution of create when $a.reads = 0$ and no events have occurred. The execution of `put_request` neither changes the number of enter and exit events or $a.reads$. The execution of `put_exit` when e is a read decreases $a.read$ by 1 (line 8) but also increases $\#x[\text{read}]$ by one. If e is a write then none of the values pertinent to (3) are changed. Thus if (3) is true prior to execution of `put_exit`, it will be true afterwards.² The execution of the operation strategy can increment $a.reads$ (line 16) but only by increasing $\#e[\text{read}]$ (line 17), assuming $a.frq$ contains only read requests. Also $\#e[\text{read}]$ can be increased only if $a.reads$ is incremented, assuming that $a.wq$ contains only write requests. Our assumptions about $a.frq$ and $a.wq$ can be proved by showing that

$$(8) \quad (a.rq) \cup (a.frq) \subseteq W(\alpha, \text{read})$$

$$(9) \quad (a.wq) \subseteq W(\alpha, \text{write})$$

are behavioral invariants. A quick examination of `rw` shows this to be so. Thus (3) is a behavioral invariant. In a similar way it can be shown that (4) is also a behavioral invariant.

1. The expression $(a.rq) \cup (a.frq)$ is used to denote the set of all requests in either $a.rq$ or $a.frq$.

2. Note that this could have been done much more formally by using the standard techniques used by Floyd for proving invariant assertions.

Now (2) is certainly true immediately after the execution of the create when $a.reads$ and $a.writes$ are both zero. So we need only show that if (2) holds prior to the execution of strategy, it will hold afterwards. This is easily done using the standard techniques used by Floyd in proving invariant assertions. Therefore, the protector defined by rw is a solution to $Mx(r[write], r')$.

At this point we turn our attention to $Fr(r)$ which has been rewritten in terms of the protector's state in (7). First we must establish that (5) and (6) are behavioral invariants in order to show that (7) is really the same as $Fr(r)$. That (6) is a behavioral invariant is easy to verify. Part of (5) is also easy (see (8) above). It is more difficult, however, to show that every time a request is removed from $a.rq$ or $a.frq$ a read is allowed to enter. This is because if $a.frq$ is not empty when lines 11-12 are executed, implicit removal of requests occurs. If we can show that

$$(10) \quad (a.writes = 0) \vee (\text{empty}(a.frq))$$

is a behavioral invariant, then we will know that lines 11-12 will never be executed unless $a.frq$ is empty. Verifying that (10) is a behavioral invariant is again straightforward.¹ Since (10) is a behavioral invariant, we know that every time a request is removed from $a.rq$ or $a.frq$ a read is allowed to enter. Thus (4) is a behavioral invariant.

Finally we are in the position to prove that (7) is true. This will be done by first proving that every request put on $a.frq$ is eventually removed and returned by strategy. Then it will be shown that every request on $a.wq$ is eventually removed and allowed to continue. Last to be shown is that if $a.rq$ is ever non-empty, eventually all requests on $a.rq$ will be put on $a.frq$.

1. In light of the truth of (10) the predicate in the test in line 13 is redundant.

Let us suppose that e is the first request on $a.frq$. Note that e can be removed only on line 15 in strategy. Now from the semantics of protectors and the fact that each of the operations of the cluster rw always terminates,¹ we know that eventually strategy will be called and e will still be the first request on $a.frq$. Thus when the test at line 13 is executed, the branch (14-17) will be taken and e will be removed and returned to the protector. Thus by simple induction on $event_seq$ it is possible to show that any event in $a.frq$ will eventually be removed and returned by strategy.

Next we examine $a.wq$. Suppose that e is the first event on $a.wq$. Now there are two cases: i) $a.writes=0$; or ii) $a.writes=1$. In the second case, the write which is in the data base will eventually leave the data base and $a.write$ will be set to zero; then we will be in the first case. From this point on no request will be added to $a.frq$ until either $a.wq$ is empty (line 3) or another write leaves the data base (line 10). In either case no requests will be added to $a.frq$ until e is removed from $a.wq$. Previously we showed that every request must eventually be removed from $a.frq$. Also we assumed that every request would return from the data base. Therefore, eventually strategy will be called with $a.reads=0$ and $a.frq$ empty and with e the first event on $a.wq$. At this point the `elseif` branch (line 18) will be true and lines 19-22 will be executed resulting in the removal of e from $a.wq$. Again by induction, it can be shown that if e is ever in $a.wq$, eventually it will be removed.

Last of all we look at $a.rq$. Requests are removed from $a.rq$ only on lines 11-12 and all are placed on $a.frq$ which, from previous arguments, implies they will eventually be returned by strategy. Thus we need only show that if a request is ever added to $a.rq$, eventually lines 11-12 will be executed. Requests are put on $a.rq$ only on line 4, when either $a.wq$ is not empty or $a.writes=1$. In any case either $a.writes=1$ now, or $a.writes$ will be equal to one in the future.² We assumed that a write in the

1. Since none of the operations contains any loops, this is immediate.

2. We know this from our previous arguments about $a.wq$

AD-A091 015 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/6 9/2

A SEMANTICS OF SYNCHRONIZATION.(U)

SEP 80 C R SEAQUIST

MIT/LCS/TM-176

N00014-75-C-0661

NL

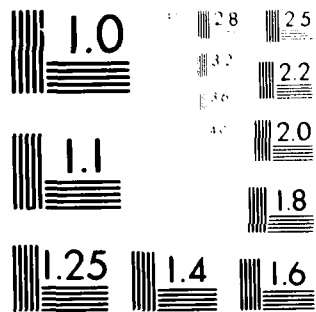
UNCLASSIFIED

2 of 2

200
C/1000



END
DATE
FILMED
11-80
DTIC



MICROCOPY RESOLUTION TEST CHART
 NATIONAL BUREAU OF STANDARDS-1963-A

data base will always eventually exit the data base. Thus eventually an exit event will be passed to put_exit and lines 11-12 will be executed. Thus (7) is true. Therefore, we have established that the protector defined by rw in figure 4.6.2.19 is a solution to (0).

4.6.3 Correctness of a Bounded Buffer Solution

In this sub-section we show that the protector defined by the cluster bb given in figure 4.6.3.20 is a solution¹ to the predicate

$$(0) \quad Z \wedge M \wedge Mx(\text{produce}, \text{produce}) \wedge Mx(\text{consume}, \text{consume}),$$

where

$$\begin{aligned} Z(\alpha) &\equiv \forall \beta [\beta < \alpha \rightarrow (0 \leq \#_{\beta} x[\text{produce}] - \#_{\beta} e[\text{consume}])] \\ M(\alpha) &\equiv \forall \beta [\beta < \alpha \rightarrow (\#_{\beta} e[\text{produce}] - \#_{\beta} x[\text{consume}] \leq \text{max})]. \end{aligned}$$

We will use a function $i: \text{Bool} \rightarrow \text{integers}$ to simplify the expression of some of the predicates used below. We define

$$\begin{aligned} i(\text{true}) &= 1, \text{ and} \\ i(\text{false}) &= 0. \end{aligned}$$

After some examination of the definition of the cluster bb, it is obvious that a.count is supposed to represent the number of items currently in the buffer while a.p_busy and a.c_busy indicate whether there are any producers or consumers, respectively, accessing the buffers. We can make this explicit by proving the following to be behavioral invariants.

- (1) $\text{a.count} = \# x[\text{produce}] - \# x[\text{consume}]$
- (2) $i(\text{a.p_busy}) = \# e[\text{produce}] - \# x[\text{produce}]$

1. This solution was discussed earlier in 4.5.

Fig. 4.6.3.20. *Solution to the Bounded Buffer Problem*

```

bb = cluster[max:int] is create, put_request, put_exit, strategy
  produce = 1
  consume = 2
  rep = record[pq:cq:event_seq.count:int,p_busy,c_busy:bool]

  create = proc() returns(cvt)
1    return(rep${pq:event_seq$create(),cq:event_seq$create(),
                  count:0,p_busy:false,c_busy:false})
    end create

  put_request = proc(e:event,a:cvt)
2    if event$(p(c)) = produce then event_seq$(a.pq,e)
3    elseif event$(p(c)) = consume then event_seq$(a.cq,e) end %if
    end put_request

  put_exit = proc(e:event,a:cvt)
4    if event$(p(c)) = produce
5    then a.count := a.count + 1
6    a.p_busy := false
7    elseif event$(p(c)) = consume
8    then a.count := a.count - 1
9    a.c_busy := false
    end %if
    end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
10   if ~a.c_busy & (0 < a.count) & ~event_seq$empty(a.cq)
11   then c:event := event_seq$frst(a.cq)
12   event_seq$dq(a.cq)
13   a.c_busy := true
14   return(c)
15   elseif ~a.p_busy & (a.count < max) & ~event_seq$empty(a.pq)
16   then c:event := event_seq$frst(a.pq)
17   event_seq$dq(a.pq)
18   a.p_busy := true
19   return(c)
    end %if
20   signal null_event
    end strategy
end bb

```

$$(3) \quad \neg(a.c_busy) = \#e[\text{consume}] - \#x[\text{consume}]$$

Showing that (1) is a behavioral invariant is straightforward. To show that (2) and (3) are behavioral invariants we must first show that

$$(4) \quad a.pq = W(\alpha, \text{produce})$$

$$(5) \quad a.cq = W(\alpha, \text{consume})$$

which is easily done. Note that by establishing the truth of (2) and (3), we show that the protector is a solution to

$$Mx(\text{produce}, \text{produce}) \wedge Mx(\text{consume}, \text{consume}).$$

Thus we need only show that the protector is a solution to $Z \wedge M$ and we will be finished. From theorem 3.4.4 we know it is sufficient to show that the following two predicates are true after the create operation and are invariants of the execution of strategy,

$$(6) \quad (0 \leq \#x[\text{produce}] - \#e[\text{consume}])$$

$$(7) \quad (\#e[\text{produce}] - \#x[\text{consume}]) \leq \max.$$

Using (1)-(3) and simple algebra, we can rewrite (6) in terms of the protector's state as follows,

$$(8) \quad 0 \leq a.\text{count} - \neg(a.c_busy).$$

Similarly we can rewrite (7) as

$$(9) \quad \neg(a.p_busy) + a.\text{count} \leq \max.$$

After create, (8) and (9) are certainly true. Now we need only show that if (8) and (9) are true before the execution of strategy, they will be true afterwards. This is easily seen to be the case. Therefore, bb defines a protector which is a solution to (0).

4.6.4 Remarks on the Methodology

In this section our understanding of the semantics of protectors has been applied in order to develop a method for proving the correctness of protectors. The examples of correctness proofs, although somewhat tedious, demonstrate that the method actually does lead to a more thorough understanding of the implementation. For instance, the fact that the conjunction in the if statement of the cluster rw (line 13, figure 4.6.2.19) was redundant was certainly not obvious when the code was first written. Thus in addition to being used to verify correctness, the understanding gained through the proof of behavioral invariants can be used to modify and optimize the code. This section has placed all its emphasis on proving that a solution satisfies consistency, priority and fairness constraints. It is, however, also possible to prove that a solution to a simple predicate is optimal. The discussion about optimal solutions in chapter 3 suggests an approach.

From theorem 3.4.5 we know that to show that a protector is an optimal solution to a simple predicate P , we must prove that whenever the operation strategy signals `null_event`, then either there are no outstanding requests, or returning a request would violate P . As an example, consider the weak readers priority solution to the readers/writers given in figure 4.6.4.21.¹ Assume that we have already proved that `a.q` actually contains all the outstanding requests and that the following are behavioral invariants:

- (0) $a.reads = \#e[read] - \#x[read]$
- (1) $a.writes = \#e[write] - \#x[write]$
- (2) $[(a.reads = 0) \vee (a.writes = 0)] \wedge [a.writes \leq 1]$.

We want to prove that whenever strategy signals `null_event`, then returning any outstanding request would violate one of (0-2). Now `null_event` can be signaled only

1. This solution was discussed previously in 4.2.

Fig. 4.6.4.21. Weak Readers Priority Solution to the Readers/Writers Problem

```

rw = cluster is create, put_request, put_exit, strategy
  read = 1
  write = 2
  rep = record[q:event_seq,writes,reads:int]

  create = proc() returns(cvt)
1    return(rep[q:event_seq$create(),writes:0,reads:0])
    end create

  put_request = proc(c:event,a:cvt)
2    event_seq$enq(a,q,c)
    end put_request

  put_exit = proc(c:event,a:cvt)
3    if event$op(c) = write then a.writes := a.writes + 1
4    elseif event$op(c) = read then a.reads := a.reads + 1 end
    end put_exit

  strategy = proc(a:cvt) returns(event) signals(null_event)
5    if (a.reads = 0) & (a.writes = 0)
6      then c:event := event_seq$first(a,q)
7      if event$op(c) = write then a.writes := a.writes + 1
8      elseif event$op(c) = read then a.reads := a.reads + 1 end
9      event_seq$deq(a,q)
10     return(c)
11   elseif a.writes = 0
12     then c:event := event_seq$firstp(a,q,readers)
13     a.reads := a.reads + 1
14     event_seq$remove(c,a,q)
15     return(c)
    end %if
16   except when empty: signal null_event end
17   signal null_event
  end strategy

  readers = proc(c:event) returns(bool)
    return(event$op(c) = read)
  end readers

end rw

```

from line 16 and line 17. If strategy signals `null_event` at line 17, then `a.writes=1` in which case no request can be returned without violating one of (0-2). If strategy signals `null_event` at line 16, there are two possibilities: i) there are no outstanding requests; or ii) there are no outstanding read requests and `a.reads>0` in which case no write requests can be returned. Thus whenever `null_event` is signaled, either there are no outstanding requests, or returning a request would violate one of (0-2). Therefore, the cluster `rw` defines a protector which implements an optimal solution to $Mx(r[write],r')$.

4.7 Conclusions

This chapter has demonstrated how the notion of a simple polling guardian can lead to a useful and practical mechanism for implementing solutions to synchronization problems. Below, the advantages of this approach are discussed.

The variety of solutions which were implemented in the previous sections testifies to the expressiveness of the mechanism. All the implementations are direct and fairly simple to follow. For example, in the fair solution to the dining philosophers problem, it was straightforward to implement Dijkstra's suggestion of having very hungry philosophers. One can easily imagine the complexity of the equivalent implementation using semaphores.

Synchronization types are easy to write. This stems partly from the ability to separate the implementation of priority constraints from the implementation of consistency constraints. The ability to use many different data types for maintaining the currently outstanding requests makes implementing priority constraints relatively easy. For example, suppose that we wished to implement a solution to

$$(0) \quad Mx(r[write],r') \wedge LIFO(r).$$

where $LIFO(r)$ is a predicate stating that all outstanding requests must be serviced in a last come first served manner. Such an implementation can be readily obtained from the FCFS solution given in figure 4.2.6, simply by changing the type of `q` (which contains the outstanding requests) from `event_seq` to a type called `event_stack`. The

operations of `event_seq`: `nq`, `frst`, and `dq` would be replaced by the corresponding operations for `event_stack`: `push`, `top`, and `pop`. Although it is probably possible to implement a solution to (0) with monitors, the implementation would undoubtedly be quite complicated and hard to understand. The complexity of the monitor solution results from the inflexibility of condition queues. The above example also suggests that it is easy to modify a synchronization type in order to conform with small changes in the specifications. As another example, recall from 4.2 how the implementations of the solutions to the readers priority and writers priority versions of the readers/writers problem were very similar. Once the readers priority version (figure 4.2.8) was written, it was simple to modify it to obtain the writers priority version (figure 4.2.9). Thus it is seen that protectors are easily modified in order to meet changes in specification.

Protectors support modularity in several important ways. For example, when a synchronization type is implemented by a particular cluster, it can be used to define many separate protectors each coordinating the accesses to distinct sets of procedures. Thus a synchronization type encapsulates a certain guardian behavior. The cluster implementing the synchronization type can be viewed as implementing a "synchronization abstraction." The use of a "synchronization abstraction" in a protector-create statement is completely separate from its implementation. This is quite different from both monitors and serializers where a new monitor or serializer must be written for each new application.

One of the criticisms often leveled at monitors is that they do not actually isolate the resource. Thus if a monitor is used with no additional structure, correct results depend on proper invocation of the monitor operations before and after each access. Unsynchronized access is, however, still possible. Protectors do not have this problem because all invocations are automatically modified in order to guarantee that only synchronized accesses occur.

In the previous section the verification of correctness was discussed. Because the semantics of the protector is based on the simple guardian, many of the theorems of Chapter 3, which help characterize the nature of solutions, can be used in the verification of correctness. The proving of behavioral invariants, in addition to being useful for verification of correctness, also helps increase understanding of an implementation and occasionally points out optimizations which can be made. For example, in proving the correctness of the fair solution to the readers priority version of the readers/writers problem, it was discovered that the predicate of a test was redundant. Thus this test could be simplified. Therefore, we see that an advantage of protectors is that they have a precise, yet simple, semantics. This helps in reasoning about implementations.

Although the verification methods discussed in the previous section are important, it is also important to be able to debug programs through the actual execution of code. Typically, debugging multi-processing systems is very difficult because it is often practically impossible to force a certain interleaving of events to occur in order to see the implementation's response. Thus even though a solution might supposedly be designed to handle a certain situation, that situation might never occur during testing. Even when a bug is found, it is usually impossible to recreate the problem. With the protector approach this is not the case. For example, suppose we had a synchronization type *s* which we wanted to test. It is a simple matter to code the protector procedure of figure 4.1.2.4 with the following changes: Replace the *in_queue* by an input stream from a file; replace *synchronization* by *s*; remove the *allow* command; and add code to log all events on a second file. (See figure 4.7.22.) Now any possible interleaving of events can be written to the input file. The procedure *test_s* can then be invoked. After the return from *test_s*, the output file needs only to be examined to see if the synchronization type *s* is actually handling the test situations correctly. Thus it is quit simple to test synchronization strategies.

Fig. 4.7.22. A Procedure for Testing a Synchronization Type s

```
test_s = proc()
  % code for opening the input file and log file should go here.
  alpha:s:=s$create()
  while true do
    c:event:=event_stream$get(Input_File)
    event_stream$put(Log_File,e)
    if event$type_of(c)="request"
      then s$put_request(c,alpha)
    elseif event$type_of(c)="exit"
      then s$put_exit(c,alpha)
    end %if
    c:=s$strategy(alpha)
    except when null_event: continue end
    event_stream$put(Log_File,c)
    allow(c)
    end %while
    except when end_of_file: end
  end test_s
```

5. Summary and Directions for Future Work

This chapter summarizes the contributions of the thesis and suggests areas for further research. There are two sections. The first reviews how the polling guardian has provided a framework for understanding synchronization. The second section addresses areas in which further work would be useful.

5.1 Summary

The main contribution of this paper is the notion of a simple polling guardian. The polling guardian has provided a framework for the discussion of synchronization problems and their solutions. Within this framework it has been possible to express solutions, examine behavior of solutions, define "good" or optimal solutions, characterize solutions to certain classes of predicates, and derive an approach for implementing solutions in actual computer systems.

By using polling guardians to describe solutions derived from standard synchronization constructs, we were able to discover certain idiosyncratic behavior which is implicit in these mechanisms. For example, when a monitor solution to the readers/writers problem was examined, it was found that on certain occasions, monitors ignore processes which are trying to exit from the resource. When the monitor solution was stated in terms of a polling guardian, the cause of and remedy for this undesirable behavior was self evident: The problem occurred because the output predicate was not the constant **true**; and the remedy was simply to make the output predicate **true**.

Defining the simple polling guardian as a functional strategy for a player of a game had important consequences. When a simple polling guardian was seen solely as a functional strategy, it was possible to prove several useful theorems. These theorems are interesting because they show that a class of predicates (i.e., the simple predicates) always have optimal solutions. The definition of simple predicates can, therefore, be

used as a guide in writing reasonable specifications. Other theorems characterize the solutions of simple predicates. These theorems are useful in checking whether or not a particular guardian is indeed a solution to a specification.

A "good" or optimal solution to a specification was defined to be a solution which always tries to keep the resource as busy as possible. It was easy to formalize this definition in terms of the simple polling guardian. By stating that a solution to a specification P must be optimal, the pathological solutions are eliminated. Among the solutions eliminated are those which do not allow concurrency when they can.

In the development of an actual synchronization mechanism, the framework provided by earlier chapters was extremely useful. We were able to define the semantics of the protector directly as a simple polling guardian. This is in direct contrast with the usual method of defining new synchronization constructs which consists of implementing them in terms of older, presumably well understood, constructs. Unfortunately, the older construct often is *not* understood. That the protector is given a simple non-operational semantics has been useful. The theorems characterizing solutions are immediately applicable to verifying the correctness of implementations because the semantics of the protector is based directly on the simple polling guardian.

The main advantage of the protector approach to synchronization is that it separates the act of using a synchronization strategy from its implementation. Thus if a programmer is writing a module implementing the procedures p_1, \dots, p_n and he decides that the accesses to these procedures should be protected, he can write

create protector for procedures: p_1, \dots, p_n using dt ,
and give a specification for the synchronization type dt . The specification can be written as a predicate on event sequences. Later a cluster may be written which implements a type that satisfies the specification. The distinction between a synchronization strategy, as embodied in the synchronization type, and its implementation eases program development, modification, and maintenance.

In conclusion, the paper has provided some insight into the problems which arise in attempting to synchronize accesses to a resource. It is hoped that, in addition, the protector approach to synchronization will be of use to programmers in that it provides a structured framework for solving complex synchronization problems.

5.2 Future Research

There should be a formal language for expressing predicates on event sequences. This language should be defined in such a way that the expressible predicates are precisely those which have certain useful properties (e.g., continuity).

Better methods are needed for verifying solutions to fairness properties. A possible approach is to examine all exclusion constraints P for which there exists a priority constraint Q such that all optimal solutions to $P \wedge Q$ are fair. For example, most interesting exclusion constraints P probably have the following properties:

- i) P is simple; and
- ii) For every history α , $P(\alpha) \wedge (B(\alpha) = \emptyset) \wedge (r \in W(\alpha)) \rightarrow P(\alpha \| e(r))$.

For such a predicate P it can be shown that any optimal solution of $P \wedge \text{FIFO}(r)$ must also be a solution to $\text{Fr}(r)$. Note that verifying a guardian G as an optimal solution to $P \wedge \text{FIFO}(r)$ is very likely to be easier than directly verifying that G is a solution to $\text{Fr}(r)$.

In describing protectors, we have stated that an object of the synchronization type encodes the important aspects of the past history. An interesting question is how much information about the past history must be encoded in order to solve a given synchronization problem. Being able to answer this question would provide a space complexity measure of the synchronization problem. Another similar question asks what is the necessary time complexity of the operations `put_request`, `put_enter`, and `strategy` to solve a given synchronization problem. As synchronization problems become more complicated, the answers to the above questions will become more

important.

In many of the synchronization problems there seems to be a trade-off between the "responsiveness" and the "throughput" of a solution. For example in the readers/writers problem, the FCFS solution (figure 4.2.6) is very "responsive" to requesters but has much less "throughput" than the readers priority solution (figure 4.2.8) which can starve requesters. The fair solution (figure 4.2.10) is not as "responsive" as the FCFS solution but it has better "throughput." When compared to the readers priority solution it is seen that the fair solution is more "responsive" but has worse "throughput." It would be interesting to be able to formalize the notions of "responsiveness" and "throughput" and to be able to make explicit their relationship given any predicate.

A generalization of the protector mechanism that could handle synchronization problems which fall outside the resource guardian model would be useful. A way to proceed is suggested by a careful examination of the request, enter, and exit events when viewed as messages sent to and from a polling guardian. A request event is a message that requires a response from the polling guardian. In the resource guardian model this response is an enter message. Thus enter events are a special case of the general class of response events. An exit event can be thought of a message sent to the guardian for which no response is necessary. More generally an exit event can be thought of as just a notice sent to the guardian. Note that every response statement must be preceded by a corresponding request statement but that notice events can occur independently. Now in the more general framework of sequences of request, response, and notice events, it is a simple matter to extend the protector mechanism to handle more general problems.

One complaint that can be made about the protector mechanism is that for a particular solution it concentrates all the duties of synchronization into a single and possibly vulnerable process. Thus the mechanism is not suitable for providing a robust implementation of a solution in a distributed system. One possible approach to solving

this problem is to have n separate protectors guarding a set of procedures instead of just one. In such a case, before an activity could invoke a protected procedure, it would put requests on the input queues of all n of the protectors. The process would then wait until all n protectors had re-activated it. If any of the protectors failed, the other protectors would detect this via a time-out mechanism. The detection of a failure would cause some sort of clean-up of the failing protector and possibly the creation of a replacement. For such an approach to work, the progress of each protector would have to keep pace with the others. Although the sketch of this approach barely hints at the problems inherent in it, preliminary study suggests that the approach is worth further investigation.

6. References

- [ANDR79] Andrews, G.R., "Synchronizing Resources," Computer Science TR 78-360, Department of Computer Science, Cornell University, Ithica, NY, Feb. 1979.
- [BLOO79] Bloom, T., "Synchronization Mechanisms for Modular Programming Languages," TR-211, Laboratory for Computer Science, M.I.T., Cambridge, Mass., Jan., 1979.
- [BURS74] Burstall, R.M., "Program Proving as Hand Simulation with a Little Induction," *Information Processing*, North Holland Publishing Company, Amsterdam, 1974, pp308-312.
- [COUR71] Courotis, P.J., F. Heymans, and D.L. Parnas, "Concurrent Control with 'Readers' and 'Writers'," *Communications of the ACM*, (14,10) Oct. 1971, pp667-668.
- [DEV177] Devillers, R., "Game Interpretation of the Deadlock Avoidance Problem," *Communications of the ACM*, (20,10) Oct., 1977, pp741-745.
- [DIJK68] Dijkstra, E.W., "Cooperating Sequential Processes," *Programming Languages* (F. Genuys, ed.), Academic Press, NY 1968, pp43-111.
- [DIJK71] Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, (1,2) 1971, pp115-138.
- [DIJK76] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [FLOY67] Floyd, R.W., "Assigning Meanings to Programs," *Mathematical Aspects of Computer Science* (J.T. Schwartz, ed.), *Proc. Symp. App. Math.*, 19, American Mathematical Society, 1967, pp19-32.
- [GALE53] Gale, D. and F.M. Stewart, "Infinite games with perfect information," *Contributions to the Theory of Games*, Annals of Mathematics Studies No. 28, Princeton University Press, 1953, pp245-266.

- [GREI75] Greif, I., "Semantics of Communicating Parallel Processes," TR-154, Project MAC, Laboratory for Computer Science, M.I.T., Cambridge, Mass., Sept. 1975.
- [GREI76] Greif, I., "Formal Problem Specification for Readers and Writers Scheduling," *Proc. MRI Symp. on Software Eng.*, Polytechnic Institute of New York, 1976, pp225-238.
- [GREI77] Greif, I., "A Language for Formal Problem Specification," *Communications of the ACM*, (20,12) Dec. 1977, pp931-935.
- [HABE72] Habermann, A.N., "Synchronization of Communicating Processes," *Communications of the ACM*, (15,3) Mar. 1972, pp171-176.
- [HABE76] Habermann, A.N., "Review of Article by Leon Presser on Multiprogramming Coordination," *Computing Reviews*, (29,788) April 1976, pp150-151.
- [HANS78] Hansen, P.B., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, (21,11) Nov. 1978, pp934-941.
- [HEWI77] Hewitt, C.E., "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence*, (8,3) June 1977, pp323-364.
- [HEWI79a] Hewitt, C.E., and R.R. Atkinson, "Specifications and Proof Techniques for Serializers," *IEEE Transactions on Software Engineering*, (SE-5,1) Jan. 1979, pp10-23.
- [HEWI79b] Hewitt, C.E., G. Attradi, and H. Lieberman, "Specifying and Proving Properties of Guardians for Distributed Systems," A.I. Memo 505, Artificial Intelligence Laboratory, M.I.T., Cambridge, Mass., June 1979.
- [HINM79] Hinman, P.G., "Borel Determinacy," *The American Mathematical Monthly*, (86,2) Feb. 1979, pp114-115.
- [HOAR74] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM*, (17,10) Oct. 1974, pp549-557.

- [KAPU80] Kapur, D.K., "Towards a Theory for Abstract Data Types," TR-237, Laboratory for Computer Science, M.I.T., Cambridge, Mass., 1980.
- [KELL76] Keller, R.M., "Formal Verification of Parallel Programs," *Communications of the ACM*, (19,7) July 1976, pp371-384.
- [KWON78] Kwong, Y.S., "Livlocks in Parallel Programs, Part I," CS-TR-78-CS-15, McMaster University, Hamilton, Ontario, August 1978.
- [LADN79] Ladner, R.E., "The Complexity of Problems in Systems of Communicating Sequential Processes," TR 79-03-01, Department of Computer Science, University of Washington, 1979.
- [LAMP78] Lamport, L., "The 'Hoare Logic' of Concurrent Programs-Preliminary Draft," CSL-79, SRI International, Nov. 1978.
- [LAVE78] Laventhal, M.S., "Synthesis of Synchronization Code for Data Abstractions," TR-203, Laboratory for Computer Science, M.I.T., Cambridge, Mass., June 1978.
- [LIND74] Lindsey, C.H., "Partial Parameterization," *ALGOL Bulletin*, No. 37, pp24-26.
- [LIND76] Lindsey, C.H., "Specification of Partial Parameterization Proposal," *ALGOL Bulletin*, No. 39, pp6-9.
- [LISK79] Liskov, B., R. Atkinson, T. Bloom, E.B. Moss, C. Schaffert, B. Scheifler, A. Snyder, "CLU Reference Manual," TR-225, Laboratory for Computer Science, M.I.T., Cambridge, Mass., Oct. 1979.
- [MANN78] Manna, Z. and R. Waldinger, "Is 'Sometimes' Sometimes Better than 'Always'? Intermittent Assertions in Proving Program Correctness," *Communications of the ACM*, (21,2) Feb. 1978, pp159-177.
- [McCA65] McCarthy, J., P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin, *LISP 1.5 Programmer's Manual*, M.I.T. Press, Cambridge, Mass., 1965.
- [PRES75] Presser, L., "Multiprogramming Coordination," *Computing Surveys*, (7,1) March 1975, pp21-44.

- [REED79] Reed, D.P. and R.K. Kanodia, "Synchronization with Events and Sequencers," *Communications of the ACM*, (22,2) Feb. 1979, pp115-123.
- [REIF80] Reif J.H. and G.L. Peterson, "A Dynamic Logic of Multiprocessing with Incomplete Information," Seventh Annual Symposium on Principles of Prog. Lang., 1980, pp193-202.
- [SCHW78] Schwarz, J.S., "Distributed Synchronization of Communicating Sequential Processes," Manuscript, University of Edinburgh, July 1978.
- [STAR79] Stark, E.W., "Semaphore Primitives and Fair Mutual Exclusion," TM-158, Laboratory for Computer Science, M.I.T., Cambridge, Mass., 1979.
- [TEOR72] Teorey, T.J. and T.B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, (15,3) March 1972, pp177-184.

OFFICIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office/Boston
Building 114, Section D
666 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office/Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office/Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D. C. 20375
6 copies

Assistant Chief for Technology
Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Naval Ocean Systems Center, Code 91
Headquarters-Computer Sciences &
Simulation Department
San Diego, CA 92152
Mr. Lloyd Z. Maudlin
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation & Math Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper, USNR
NAVDAC-OOH
Department of the Navy
Washington, D. C. 20374
1 copy